

AD-A195 520

TAC-1: A KNOWLEDGE-BASED AIR FORCE TACTICAL BATTLE

1/1

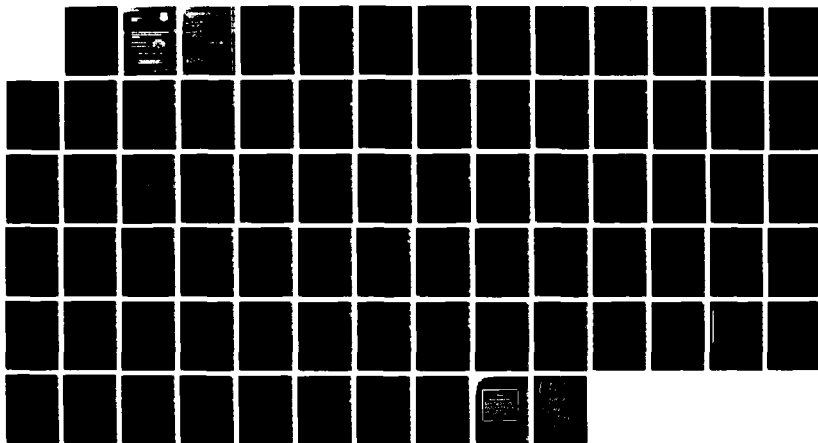
MANAGEMENT TESTBED(U) MITRE CORP MCLEAN VA

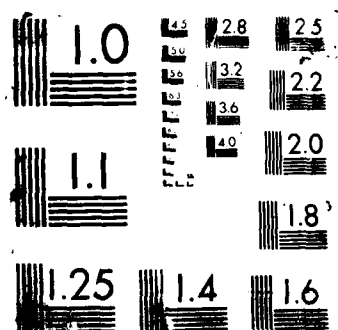
R O NUGENT ET AL. JAN 88 RADC-TR-88-10 F19628-87-C-0001

F/G 12/5

NL

UNCLASSIFIED





AD-A195 520

**TAC-1: A KNOWLEDGE-BASED AIR FORCE
TACTICAL BATTLE MANAGEMENT
TESTBED**

THE WIRE CORPORATION

ROBERT D. HUGHES AND ROBERT W. TERRY

**DTIC
ELECTE
NOV 17 1980**

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED

**ROME AIR DEVELOPMENT CENTER
Air Force Systems Command
GRANDBAY AIR FORCE BASE, NY 13441-5700**

88 5 16 06 8

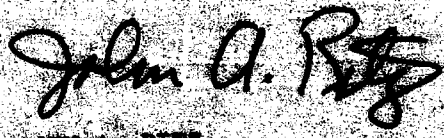
has been reviewed by the RADC Public Affairs Office (PA) and
the National Technical Information Service (NTIS). At NTIS
it is available to the general public, including foreign nations.

It has been reviewed and is approved for publication.


DONALD F. ROBERTS
Project Engineer


RAYMOND F. ORTIZ, JR.
Technical Director
Directorate of Command & Control

FOR THE COMMANDER:


JOHN A. RITZ
Directorate of Plans & Programs

If your address has changed or if you wish to be removed from the RADC
mailing list, or if the addressee is no longer employed by your organization,
please notify RADC (COMB) Griffiss AFB NY 13441-5700. This will assist us in
maintaining a current mailing list.

Do not return copies of this report unless contractual obligations or
notices on a specific document require that it be returned.

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE				Form Approved OMB No. 0704-0188		
1a. REPORT SECURITY CLASSIFICATION UNCLASSIFIED			1b. RESTRICTIVE MARKINGS N/A			
2a. SECURITY CLASSIFICATION AUTHORITY N/A			3. DISTRIBUTION/AVAILABILITY OF REPORT Approved for public release; distribution unlimited.			
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE N/A						
4. PERFORMING ORGANIZATION REPORT NUMBER(S) N/A			5. MONITORING ORGANIZATION REPORT NUMBER(S) RADC-TR-88-10			
6a. NAME OF PERFORMING ORGANIZATION The MITRE Corporation		6b. OFFICE SYMBOL (If applicable)	7a. NAME OF MONITORING ORGANIZATION Rome Air Development Center (COES)			
6c. ADDRESS (City, State, and ZIP Code) 7525 Colshire Drive McLean VA 22102-3481			7b. ADDRESS (City, State, and ZIP Code) Griffiss AFB NY 13441-5700			
8a. NAME OF FUNDING/SPONSORING ORGANIZATION Rome Air Development Center		8b. OFFICE SYMBOL (If applicable) COES	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER F19628-87-C-0001			
8c. ADDRESS (City, State, and ZIP Code) Griffiss AFB NY 13441-5700			10. SOURCE OF FUNDING NUMBERS			
			PROGRAM ELEMENT NO. 62702F	PROJECT NO. 5581	TASK NO. 27	WORK UNIT ACCESSION NO. 31
11. TITLE (Include Security Classification) TAC-1: A KNOWLEDGE-BASED AIR FORCE TACTICAL BATTLE MANAGEMENT TESTBED						
12. PERSONAL AUTHOR(S) Richard O. Nugent, Richard W. Tucker						
13a. TYPE OF REPORT Interim		13b. TIME COVERED FROM Oct 86 TO Sep 87		14. DATE OF REPORT (Year, Month, Day) March 1988		
15. PAGE COUNT 84						
16. SUPPLEMENTARY NOTATION N/A						
17. COSATI CODES			18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)			
FIELD	GROUP	SUB-GROUP	Artificial Intelligence, Knowledge-Based Systems, Distributed Artificial Intelligence, Cooperating Knowledge Based Systems, Knowledge-Based Tactical Battle Management.			
09	02					
19. ABSTRACT (Continue on reverse if necessary and identify by block number) This report describes the framework for, and a demonstration vehicle of, a knowledge-based testbed for integrating multiple artificial intelligence systems into a distributed processing network for purposes of evaluation and exploitation. TAC-1 is a version of the testbed applied to the domain of Air Force tactical battle management. The domain-independent framework includes a centralized control subnet, including a message router and a common protocol language for message passing among component systems. A Common Database and a Common Knowledge Base are essential components of the testbed. The Router directs data queries to the Common Database (one of the hosted systems) and, through the use of a Common Knowledge Base, directs service requests to the systems which can handle them.						
20. DISTRIBUTION/AVAILABILITY OF ABSTRACT <input checked="" type="checkbox"/> UNCLASSIFIED/UNLIMITED <input type="checkbox"/> SAME AS RPT. <input type="checkbox"/> DTIC USERS			21. ABSTRACT SECURITY CLASSIFICATION UNCLASSIFIED			
22a. NAME OF RESPONSIBLE INDIVIDUAL Donald F. Roberts			22b. TELEPHONE (Include Area Code) (315) 330-2973		22c. OFFICE SYMBOL RADC (COES)	

DD Form 1473, JUN 86

Previous editions are obsolete.

SECURITY CLASSIFICATION OF THIS PAGE

UNCLASSIFIED

ACKNOWLEDGMENTS

Special thanks are due to Paul Morawski for laying the groundwork for the project during the project's first half-year. Many beneficial ideas were suggested by Dr. Richard H. Brown, Robert C. Labonté, Donald Roberts, R. Peter Bonasso, Dr. John W. Benoit, Karl S. Schwamb, and Christopher Elsaesser. We would like to thank Alice Schafer and Dr. Brown for providing a version of their CAMPS relational database management software for use in TAC-1.



Accession For	
NTIS CRA&I	<input checked="" type="checkbox"/>
DTIC TAB	<input type="checkbox"/>
Unannounced	<input type="checkbox"/>
Justification	
By	
Date	
Availability Codes	
Code	Availability Codes
A-1	

TABLE OF CONTENTS

	<i>Page</i>
1.0 INTRODUCTION	1-1
1.1 Background	1-1
1.2 Objectives	1-2
1.3 Testbed Characteristics	1-3
1.4 Assessment Summary	1-3
1.5 Organization of Report	1-4
2.0 TESTBED DESIGN RATIONALE	2-1
2.1 Operating Environment Issues	2-1
2.1.1 Centralized Versus Decentralized Control	2-2
2.1.2 Component Connectivity	2-2
2.1.3 Common Functionality	2-3
2.1.4 Common Language	2-4
2.1.5 Synchronization of Processing	2-4
2.1.6 Heterogeneous Hardware and Software	2-5

TABLE OF CONTENTS (continued)

	<i>Page</i>
2.2 Domain System Issues	2-6
2.2.1 KBS Decomposition	2-6
2.2.2 Dependence on Other Systems	2-6
2.2.3 Robustness	2-7
2.2.4 Well-Behaved Modularity	2-8
2.2.5 Impact on Using Existing Systems	2-8
3.0 IMPLEMENTATION	3-1
3.1 TAC-1 Components	3-1
3.1.1 Required Testbed Components	3-1
3.1.2 TAC-1 Domain Components	3-4
3.2 TAC-1 Demonstration	3-7
3.3 Prototyping Environment	3-12
3.4 Description of the KB-BATMAN Shell	3-13
3.4.1 Processes	3-13
3.4.2 Input Ports	3-17
3.4.3 Message Passing	3-18

TABLE OF CONTENTS (continued)

	<i>Page</i>
4.0 ASSESSMENT OF FIRST YEAR ACCOMPLISHMENTS	4-1
4.1 Summary of Accomplishments	4-1
4.2 Assessment of TAC-1 Implementation	4-1
5.0 FUTURE DIRECTIONS	5-1
APPENDIX A: SOFTWARE DESCRIPTION	A-1
A.1 Overview	A-1
A.2 Packages	A-1
A.3 Flavors and Methods	A-2
A.4 Creating and Destroying Components	A-3
A.5 Declarations	A-4
A.6 Defining Interfaces Between Systems and the Router	A-6
A.7 Database Access	A-12
A.7.1 New Database Access Functions	A-12
A.7.2 Remote Database Access	A-16

TABLE OF CONTENTS (concluded)

	<i>Page</i>
A.8 Common Knowledge Base	A-17
A.9 User Interface	A-17
A.10 Summary of Operation	A-20

LIST OF FIGURES

<i>Figure Number</i>		<i>Page</i>
3-1	TAC-1 COMPONENTS IN A STAR NETWORK	3-2
3-2	TAC-1 DOMAIN COMPONENTS	3-6
3-3	TAC-1 DEMONSTRATION SEQUENCE: FUNCTIONAL FLOW DIAGRAM	3-8
3-4	PROCESS CONNECTIVITY IN A TESTBED	3-15
A-1	TAC-1 DEMONSTRATION: MONITOR SCREEN	A-19

LIST OF TABLES

<i>Table Number</i>		<i>Page</i>
4-1	SUMMARY OF ASSESSMENT	4-2
A-1	RELATIONAL DATABASE FUNCTIONS	A-13

1.0 INTRODUCTION

This report describes the background, design, and implementation of TAC-1, a prototype knowledge-based battle management testbed designed for the Air Force to demonstrate the concept of integrating multiple command and control (C²) systems employing artificial intelligence (AI) technology.

A principal goal of this project is to construct a framework in which diverse knowledge-based systems (KBS) can cooperate to produce results. This framework is called the Knowledge-Based Battle Management Shell (KB-BATMAN Shell).

1.1 Background

The Rome Air Development Center (RADC) is a United States Air Force laboratory involved with the use of artificial intelligence and other advanced technologies in C² systems. Many of the applied AI systems produced for RADC are stand-alone decision aids. Recently RADC has been interested in determining the requirements which enable C² decision aids to cooperate and to share knowledge with other systems (Walter, 1986).

It is difficult to evaluate the effectiveness of a decision aid in isolation from other systems with which it may interact. For C² software tools to be evaluated realistically, they must be able to operate and be tested under conditions which include interaction with other systems and the effects of tactical staff operations. These conditions must reflect the functioning and the effects of the C² process; that is, the evaluation environment must reflect both the simultaneous operation of other C² functions and changes to the environment caused by enemy actions, other C² functional systems, and the system itself (Graham, 1983).

There is no standard methodology for integrating distributed C² systems. Current research efforts in the field of distributed artificial intelligence (DAI) are concerned with creating models for employing intelligent agents to cooperate to solve complex problems on physically separated, multiple processors (Agha, 1987; Yonezawa and Tokoro, 1987).

Previous MITRE work on the AirLand Loosely Integrated Expert Systems (ALLIES) project (Benoit et al., 1986) involved integrating a planning system, an

intelligence analysis system, and a simulation system into a single cooperating environment. Since these systems were integrated after each was developed to operate stand-alone, the methodology for integration was ad hoc and communications required several different protocols.

A better environment for developing cooperating, distributed systems is essential to encourage modularity of system design and to provide well-defined interfaces among systems. Teknowledge, Inc. is currently developing ABE (Erman, Lark, and Hayes-Roth, 1986) which is intended to meet these goals. ABE development to date has focused on creating a module-oriented programming environment and problem-solving frameworks. However, ABE has not yet addressed the issues of distributed cooperation and is not oriented specifically to the Air Force tactical domain.

The target domain of Air Force tactical battle management places additional requirements on the testbed. AI systems in support of the Tactical Air Control System (TACS) are planned to be used as decision aids; that is, the systems will normally operate interactively with a user/operator (Weber, 1987). Needs have been defined for systems which can operate in real, or near-real time, as well as systems which have long time lines. Another requirement that is evident from looking at the types and amount of data which the systems must handle and most likely share is the ability to work with large databases. Concepts for future operations change the locations where particular functions are carried out, but the functions themselves do not change considerably. One change that will have a large effect will be the emphasis on dispersed operations (Thomas et al., 1986).

1.2 Objectives

The principal objective of this project is to develop a prototype software framework for a knowledge-based battle management testbed both to evaluate single and multiple KBS's and to investigate issues of cooperation and integration of distributed C² systems.

The purpose of TAC-1 is to demonstrate the functionality of this framework and its utility in integrating, testing, and evaluating multiple Air Force decision aids in a distributed and cooperative tactical battle management environment.

1.3 Testbed Characteristics

Required testbed characteristics derive from the project objectives and the nature of the tactical domain; these requirements are summarized below:

- The testbed must provide the connectivity, communications, and control to support distributed and cooperative problem solving. The type of control used must permit loose coupling and asynchronous operation of the component systems.
- The testbed should be flexible enough to permit modelling of a wide range of C² systems, and these systems may operate in different hardware and software environments.
- The testbed should provide the capability for the individual hosted systems to share knowledge, as well as data, so they can cooperate in solving common and joint problems.
- The testbed should provide the necessary centralization of control to assist in evaluation of the hosted systems.

1.4 Assessment Summary

The following is a summary assessment of the KB-BATMAN Shell in meeting the above required testbed characteristics. A more detailed assessment is given in Section 4.

- The KB-BATMAN Shell provides the necessary connectivity, communications, and control needed to support distributed and cooperative problem solving. Both synchronous and asynchronous operations are supported, allowing the component systems to be loosely coupled.
- Through connectivity, common language, and control the KB-BATMAN Shell provides the flexibility whereby different systems may be integrated with the least amount of effort.

- The KB-BATMAN Shell provides the Common Database and Common Knowledge Base whereby hosted systems can, in their effort to cooperate, share both data and knowledge.
- The KB-BATMAN Shell forces all communications through a central process, thereby providing the required degree of control centralization.

1.5 Organization of Report

A discussion of the design considerations involved in building TAC-1 and the KB-BATMAN Shell is provided in Section 2. Section 3 describes the implementation of the testbed and the TAC-1 demonstration. Section 4 provides a summary of the results of the project and an assessment as to how well the requirements were met. Section 5 suggests future directions for improving the concepts involved. Appendix A contains more detailed descriptions of the software than are given in Section 3. A glossary of terminology and acronyms appears at the end of this report.

2.0 TESTBED DESIGN RATIONALE

This section discusses the rationale used in formulating a design for the testbed which satisfies the requirements stated in Section 1. Two different sets of design issues need to be resolved, one set having to do with the design of the operating environment, and the other applicable to hosted systems. These issues are discussed in the following sections. For each issue, alternatives are described, and reasons for choosing particular alternatives are given. Some choices were made to reduce the scope of the project in its first year; subsequent work may require rethinking of these choices. In some cases, the proper choice was not evident and was subjected to experimentation by incremental prototyping in software. Refer to Section 3 for a description of the implementation.

2.1 Operating Environment Issues

In TAC-1, component systems are likely to be executing on different computers. In order to cooperate with other systems, a system must be able to communicate with them. Systems use *messages* to transmit information to other systems. The *form, content, and path of a message* must adhere to common protocols for cooperation to be successful. In addition, the operating environment must support various mechanisms for transmitting messages.

The following issues are relevant to the design of a distributed environment for cooperating knowledge-based systems (the KB-BATMAN Shell) and are discussed in subsequent subsections:

- Centralized versus decentralized control
- Connectivity of component systems
- Common knowledge and functionality needed by most component systems
- Adoption of a common intersystem communications language
- Synchronization of processing
- Use of heterogeneous hardware and software

Efficiency is an issue that is not addressed directly by the testbed concept. Efficiency considerations should apply within each component system, within the KB-BATMAN Shell, as well as in the frequency and composition of requests that a system makes to remote components.

The following subsections discuss the above factors. See Section 3 for descriptions of what features actually have been implemented in TAC-1.

2.1.1 Centralized Versus Decentralized Control

In any distributed environment, control of intersystem activities may be centralized or decentralized, or a hybrid. With centralized control, all messages from a system are sent to a central controlling mechanism which knows how to send the message to the appropriate system. Messages must first pass through a central point before reaching a target system. With decentralized control, which was used in MITRE's ALLIES project (Benoit et al., 1986), each system has some knowledge about the responsibilities of other systems and how to direct a message toward potential receivers. Centralized control was selected for use in the Shell, primarily to simplify the connectivity of systems. Future work will need to address the issues of decentralized control, since it better matches the control mechanisms employed in typical C² environments.

2.1.2 Component Connectivity

The literature of networks, communications, and distributed systems describes a variety of ways to connect components together for communications (Gien and Zimmerman, 1979; Green, 1979). For the initial series of TAC-1 prototypes, it was decided to limit the configuration of TAC-1 components to a single network rather than a collection of networks connected by gateways. It also was decided to centralize control of the network into a subnet which would be the center of a star network of component systems. In a star network, each system maintains communications only with a central node, and the central node maintains communications with all systems. A major alternative to a star network configuration is a many-to-many connectivity in which each system maintains a communications path with each other system that it needs to communicate with. The star network approach was chosen in order to discourage systems from being designed to be too dependent on the existence of other

systems and to simplify monitoring of messages within the testbed. The principal disadvantage of a star network is the great reliance on the continuous operation of the central node. If the central node fails, no systems can communicate. Future enhancements to TAC-1 may require incorporating several network topologies into a hybrid including clusters of systems around localized control subnets.

2.1.3 Common Functionality

Even though component domain systems in TAC-1 should be loosely-coupled, as discussed in Section 2.1.5, there still is a need for sharing information that is not specific to a single system. Two common systems have been identified to fulfill this requirement: a Common Database (CDB) and a Common Knowledge Base (CKB).

A relational database management system (RDBMS) is used for the Common Database because the features, behavior, and implementation of RDBMS's are fairly standard and RDBMS implementations are available for a wide variety of computers (Date, 1987).

The Common Knowledge Base contains domain-dependent knowledge and behaviors. For TAC-1, it will contain doctrinal knowledge about Air Force tactical battle management that is general-purpose and applicable to multiple systems. Anticipating future connectivity with different echelons of Air Force and other service C², CKB could provide the link whereby knowledge is passed between echelons of command and services. The knowledge present in CKB may be fairly static and be updated solely by its associated behaviors, or it may be fairly dynamic and be updated by any system. If it is dynamic, the structure and representation of the knowledge base must be well-defined and shared by all systems that use it so no misinterpretation is likely. A similar admonition applies to the Common Database, but since knowledge representations are not standardized, the likelihood for problems is greater in CKB.

A future prospect for inclusion in the KB-BATMAN Shell is an object-oriented DBMS (Peterson, 1987). Since many of the component systems of the testbed are intended to be knowledge-based, and KBS's typically are implemented using object-oriented methodologies, such a system might be a good choice. An object-oriented DBMS may be an appropriate vehicle for storing and transmitting data and knowledge, and hence might be able to subsume the capabilities of the Common Database and Common Knowledge Base into a Common Object Base. However, a variety of different types of object-oriented DBMS's are the subjects of current research, and

it was considered not prudent to select a new style of DBMS which might evolve as research progresses.

2.1.4 Common Language

In ALLIES (Benoit et al., 1986, page 78), the lack of common semantics across systems was considered a key problem that limited the extensibility of ALLIES' loosely-integrated systems concept. A distributed set of knowledge-base systems must use a common language protocol to ensure cooperation in solving problems. If each system had to translate from its private language to the language of each system with which it might communicate, at most n^2 types of translation would be necessary in a suite of n systems. Using a common language reduces the required number of translations to at most n , one per system.

In a distributed environment, information transferred from a system to a remote system can be transmitted containing only elementary forms of data, such as numbers and character strings. A complex object such as a database relation cannot be transmitted by reference (its computer address) since the reference is meaningless on another computer. Instead, a common language for specifying how to reconstruct objects remotely is necessary for those objects which must be transmitted. The reason to transmit objects as reconstructable objects is to retain the object abstraction so remote system software does not need to know how objects are implemented. So far in the development of the KB-BATMAN Shell, database relations are the only objects transmitted that have been identified as needing reconstruction. Once reconstructed on a remote machine, a system may use the relation object locally; see Section 3 for more details on distributed access to the Common Database.

2.1.5 Synchronization of Processing

In a collection of distributed systems, coordination of the processing states of systems can be difficult due to the asynchronous execution of the systems. The systems for which the testbed is intended should be loosely coupled; in other words, each system should not depend synchronously (by waiting) on products of another system, since other systems may fail to produce those products. Toward this end, the ultimate goal in developing systems is to make them *reactive* rather than *responsive*. A responsive system would receive a message, process the message, and return a

response as a reply to the requester. A reactive system does not necessarily react to a message by sending a reply message to the system originating the message. Its reaction to a message may result in further messages being transmitted, but the originating system does not view any such incoming messages as being coupled transactionally as a reply to the original message. The requirement for the testbed to support asynchronous processing suggests the use of reactive systems, since each message can be considered to be an event that is independent of other messages.

To the extent possible, a system should be designed to operate reactively. This approach encourages loose coupling of systems. A responsive approach is desirable when synchronization of events must be ensured. For example, access to the Common Database may most appropriately be synchronous. A system requesting data from the CDB would send a request message to be executed like a remote procedure call and then wait for a reply. For the purposes of debugging interactions among systems, the synchronous remote procedure call is very useful. Due to the skeletal nature of the domain systems in TAC-1, most of the intersystem communications consist of synchronous accesses to the CDB. However, the KB-BATMAN Shell supports both responsive (synchronous) and reactive (asynchronous) methods of communication.

2.1.6 Heterogeneous Hardware and Software

Military domain KBS's are developed for operation on a wide variety of hardware and are written using various programming languages and environments, both conventional and special-purpose. Often prototype AI systems are written in a flexible development language such as LISP; for production of an operational version, such systems must be rewritten in a mandated language such as Ada.

The design for the KB-BATMAN Shell should be careful to avoid dependencies on features available only on the Symbolics LISP machines that are being used to develop the TAC-1 prototype. For example, LISP machine software environments extensively use shared memory; is shared memory a requirement for an operational KB-BATMAN Shell, or is it merely convenient for rapid prototyping of TAC-1? In any case, the development of TAC-1 (and TAC-2) may still produce requirements for the operating environment.

A goal for the testbed is to aim for the possibility of integrating systems hosted on various types of hardware and implemented in various programming languages. Some design decisions impact the requirements for the capabilities of the hardware

and software, such as requiring asynchronous operations and multitasking. In the current implementation, all software is written in LISP and executes on LISP machine hardware.

2.2 Domain System Issues

The following factors apply to the knowledge-based systems to be hosted in the testbed and are discussed in subsequent subsections:

- Optimal decomposition of KBS functionality
- Level of dependence on other systems
- Robustness of behavior under degraded conditions
- Requirements for a system to be useful as a well-behaved module in the testbed

2.2.1 KBS Decomposition

A principal issue is to determine what knowledge is generally applicable to most component systems and should be in the Common Knowledge Base.

Another issue is the level of granularity of the set of component systems, whether it is coarse-grained, with about five systems, or fine-grained, with 50 or more. The component systems are cooperating to work on multiple tasks, rather than just sub-problems of a single task. It was decided to apply the testbed to coarse-grained cooperative systems. Each system is responsible for one or more tasks. Each system may or may not be knowledge-based.

2.2.2 Dependence on Other Systems

When a system requests a service that another system can perform, two possibilities should be considered: the request can be directed to a particular system by name, or to an external controller (if any) which identifies an eligible server to handle it. If a request is sent to a particular system for execution, there is no ambiguity as

to who is responsible for handling it. However, the requester would be depending on the existence and reliability of the named target system. If other systems were capable of handling the request, the requester would ignore them, even if one were better.

If a request is to be handled by an external controller, a mechanism must exist to route the request to one or more eligible handler systems. Using centralized control, the centralized control mechanism, called a *Router* in the KB-BATMAN Shell, is responsible for determining which handler(s) should be sent the request. The Router can base its selection on information it has received from systems which declare their capabilities and interests. The choice also can be based on preference criteria relayed with the request in order to give priority to a particular handler, based on, for example, speed of handling or accuracy of results. The main advantage of using the Router is that control knowledge is centralized and therefore easier to maintain and understand than if the knowledge is distributed. The main disadvantage of the Router is that systems cannot cooperate without it. Also, using the Router becomes cumbersome and possibly inefficient when a large number of systems need to cooperate.

Some systems may wish to be notified when certain events occur. For example, if a database relation concerning enemy order of battle is changed, an intelligence system may want to be informed in order to update its situation. Events either can be announced to all systems, or announced to interested systems. In the case of the database update notification, event monitoring might be localized in each interested system, by polling, or might be procedurally attached to the database update service in the DBMS for the Common Database. The latter option was chosen for the KB-BATMAN Shell because it was considered more efficient since the DBMS has the first chance to detect any database changes. Interested systems declare their interests to the Router, which matches event notifications with event interests.

2.2.3 Robustness

In a distributed C² environment, it is important for a system to be robust in order to provide reliable services. Robustness includes the following features:

- Resistance to errors.
- Resilience when requests are not handled.

- Graceful degradation, to fail soft.

Since hosted systems are executing asynchronously in a distributed environment, each system must be able to judge whether the inputs it receives are accurate, timely, and complete for its own purposes. For example, it may be better for a planning system to use statistical averages for weather data rather than two-day-old nowcasts of meteorological data. It generally is not correct to rely on the quality of inputs unless quality indicators are provided or can be discerned by a receiving system.

Each hosted system in the testbed should be resilient to inadequate behavior of other components. Response behaviors are never guaranteed, since it is possible that no system can handle a requested service, or a server system may be too busy to handle the request in a timely fashion. The key point is that there is a general level of cooperation among the involved systems, but it is loosely coupled. When a system is designed to operate in this environment, it can assume that ordinarily a requested service will be performed, but sometimes it will not. The system should be designed to be resilient: to allow for the worst case but to plan for the best.

2.2.4 Well-Behaved Modularity

Each domain system must have a well-defined interface with other systems. If the recommended conventions for interfacing with the subnet and executing requests in systems are used, the possibility of problems should be limited. Any tendency toward degraded performance, deadlock, or component failure should be in the domain system's processes rather than in the supporting processes in the subnet.

2.2.5 Impact on Using Existing Systems

It is impractical for our testbed design to support the inclusion of any arbitrary existing decision aid system, knowledge-based or otherwise. A principal tenet of the testbed concept of distributed cooperation is to support communications among systems. However, most decision aids are designed to be stand-alone systems with only user interaction. In order to adapt a stand-alone system to be cooperative with other systems, the design of the software must be evaluated on a case-by-case basis to determine how much software will need to be modified. It is very likely that

reimplementing the system may be more cost-effective and reliable than patching existing software.

As an example, the MITRE-developed KNOBS Replanning System (KRS) system is a stand-alone KBS with a substantial user interface including support for natural language input (Tachmindji and Lafferty, 1986). An early version of the testbed used KRS as a component domain system. In order to permit KRS to accept external requests for KRS evaluation and to communicate results to an external system, inelegant software modifications were necessary. Even though it was possible to ask KRS to perform external requests, there was an ever-present possibility that an external request would clash with a user request and cause problems, since KRS' design supported only a single input thread. The user interface is tightly-coupled with the planning functions. In addition, much of the knowledge in KRS is hard-coded in software, which does not lend itself to transmission to other systems. The developers of KRS recognized these limitations, and are currently reimplementing it with enhancements as A Meta-Planning System (AMPS).

It was decided that the design of the testbed should not be aimed at supporting a wide variety of ad hoc protocols with inelegant solutions in order to support the integration of existing systems. Instead, the testbed design will include requirements for systems to be integrated and will provide conventions for systems to follow.

3.0 IMPLEMENTATION

This section describes the current implementation version of TAC-1, including the hardware and software used, a description of the features and operation of the KB-BATMAN Shell, and the component systems of TAC-1.

3.1 TAC-1 Components

TAC-1 demonstrates the use of the KB-BATMAN Shell framework to implement a simple Air Force tactical planning problem.

TAC-1 consists of six components, as shown in Figure 3-1. Three components are required in any testbed configuration: Router, Common Database, and Common Knowledge Base. TAC-1 includes the following components which implement three domain systems:

- INTEL, an knowledge-based intelligence system
- PLANNER, a knowledge-based planning system
- SIMULATOR, a knowledge-based simulator

Each of these domain systems was implemented using ERIC (Hilton, 1987). These systems have only skeletal capabilities, as discussed later, but serve to exercise and demonstrate the methods and problems of integrating, communicating, and sharing knowledge and data.

In addition to these three domain systems, domain knowledge is part of the Common Knowledge Base; it is considered a meta-level of knowledge concerning the particular domain rather than specific detailed knowledge which should be handled by a domain system responsible for a specific application role.

3.1.1 Required Testbed Components

3.1.1.1 Common Database. Distribution of information among systems occurs through a Common Database system which is available to other systems through

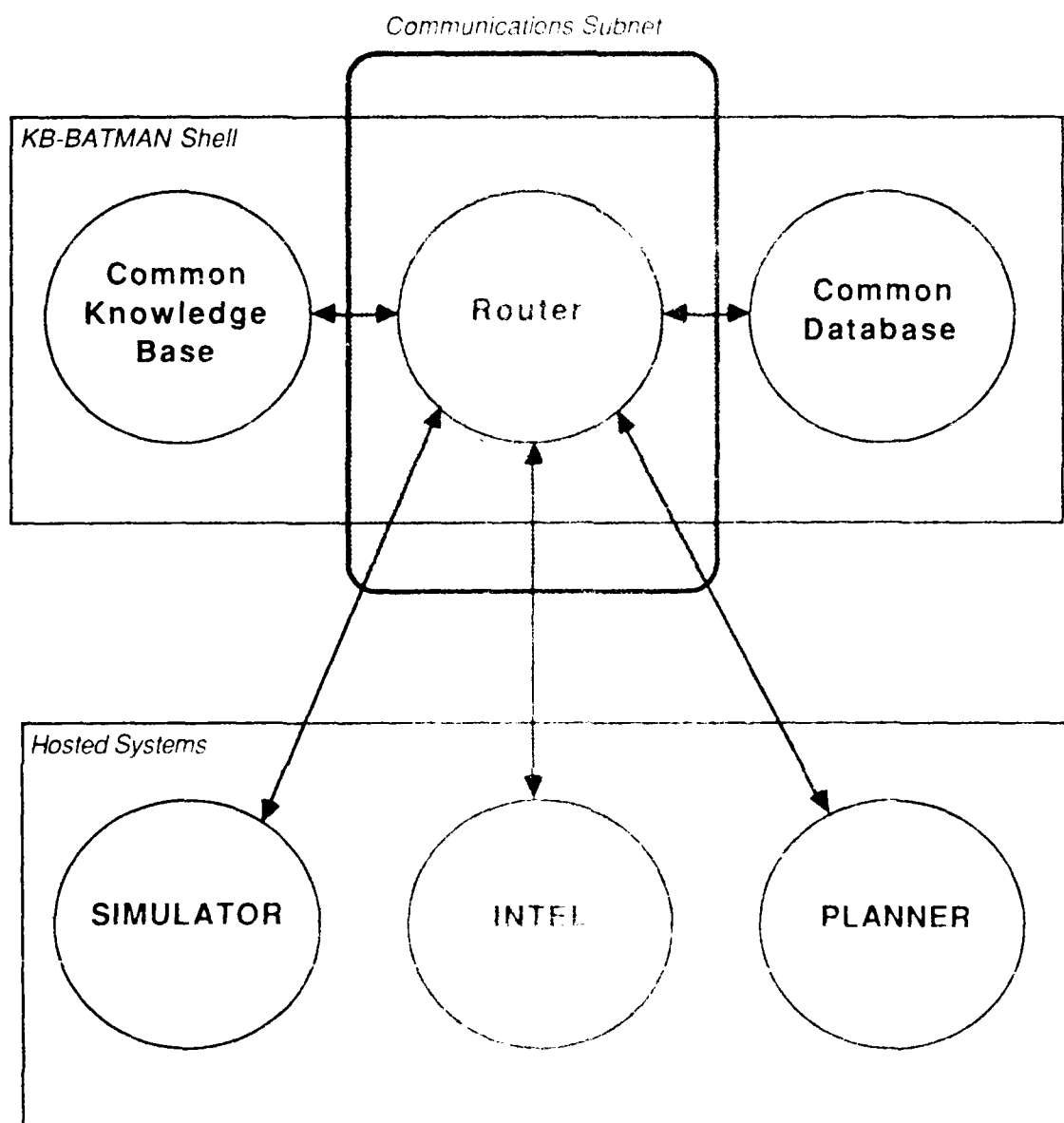


FIGURE 3-1
TAC-1 COMPONENTS IN A STAR NETWORK

the Router as a server for database services. In the present implementation, the CDB system plugs into the testbed just as any other system would; it can execute on any machine and can be replaced by any system which supports the same database services. (Issues of efficient access to distributed databases are ignored in the current implementation.) A system may declare to the Router that it wants to be notified when there is a change to the database. The accuracy, timeliness, and appropriateness of any data retrieved from the database must be determined by the system using the data or by its interface object.

The database management system (DBMS) used in the testbed is a modified version of the Core of a Meta-Planning System (CAMP'S) relational DBMS (RDBMS) (Brown and Schafer, 1987). The CAMP'S RDBMS was enhanced for use in the testbed to support distributed access in a transparent manner. In this enhanced RDBMS, relation objects are transmitted to other machines by sending a description of how to reconstruct the relations for local manipulation.

The CAMP'S RDBMS employs standard SQL-style functions including select, join, project, update, and the use of a cursor to iterate through rows of a relation object. A locally-available relation is accessed by referring to the relation's LISP object, which essentially is a pointer to the local representation for the relation data. All RDBMS functions can be used to access local database relations.

In the testbed RDBMS, the distributed access extensions permit a system to access a relation in the Common Database by referencing it by its *name* given as a character string, e.g., "REPORTED-TARGETS". All RDBMS functions except cursor functions can be used to access the CDB in this way. When a system refers to a database relation by name, the database function used performs all necessary communications and processing to produce a local copy of the relation for local manipulation. This processing includes sending a request to the subnet to perform a database function (the "DB-FUNCTION" service which is declared by the Common Database system), waiting for a reply, reading the reply, and interpreting the reply if necessary to produce a local relation object constructed from information provided in the reply message. Some RDBMS functions return data other than relations, such as value-retrieval functions (*empty-relation?*, *values-from-relation*, etc.); these values can be used without special reconstructive processing.

The system that made the database request can manipulate a local relation object without any possibility of changing relations in the CDB. The CDB can be modified by using an RDBMS modification function (*update*, *add-tuple*, *add-tuples*,

equality-delete) and citing a relation name rather than a relation object.

The testbed RDBMS supports a relation update notification service. Any system can ask to be notified when a particular relation has been modified. Whenever a modification function is used to change any such relation in the CDB, the RDBMS transmits a notification message to the subnet for the Router to direct to interested systems. The notification message consists of the name of the relation and the time updated; it does not contain any data from the relation or the type of update, and the source of the update is not provided.

In summary, the RDBMS used in TAC-1 supports a Common Database which can reside on any machine, and includes access routines for use on any machine. Access routines can manipulate relations in the CDB and also relations that are locally maintained. Local relations can be privately defined for use within a system and also can be derived from relations in the CDB.

3.1.1.2 Common Knowledge Base. The CKB system implemented in TAC-1 contains little functionality and knowledge, but is sufficient to demonstrate its intended use. The CKB system is attached to the subnet just like any other system, and can be replaced dynamically by another system which provides similar services. Currently, the knowledge of the CKB is embodied in defined behaviors; each CKB behavior is a service declared to the subnet. Sections 3.2 and A.8 describe how CKB currently is used in TAC-1.

It is anticipated that future work on the CKB will expand its role to make it a key component in the testbed. Services provided by the CKB may shadow or be shadowed by services provided by other systems, depending on the preference criteria of service requesters and the quality of the CKB's behaviors. Hence the CKB either can be considered to provide a backup role for otherwise unhandled services, or it can be considered to be an authoritative source. It has not yet been determined which knowledge-based techniques are best suited for C² battle management.

3.1.2 TAC-1 Domain Components

The domain-dependent components of TAC-1 are systems and interfaces which perform Air Force battle management functions. As no full-fledged KBS was available which met the criteria of being readily accessible, in the Air Force tactical

domain, and readily adaptable for integration in TAC-1, three emulation systems were constructed with simple functionality: INTEL, PLANNER, and SIMULATOR. The relationship of these three systems is shown in Figure 3-2. These systems are described below.

3.1.2.1 INTEL. Two functional intelligence capabilities that are critical to the operation of other nodes within the command and control network are those of maintaining the enemy air order of battle (AOB) and producing the prioritized target lists (PTL). For simplification, it will be considered that only one AOB and one PTL are produced. Inputs which are used to change the current AOB come from intelligence and operational reports (which will come from the SIMULATOR). Inputs which affect target prioritization include the target priority list (which will come from the Common Knowledge Base as an external effect, simulating input from the appropriate Air Force or joint operations center).

Input requirements for INTEL are the reported targets and the list of target priorities.

Output capabilities for INTEL are limited essentially to the prioritized target list, although the list of reported targets is probably closest to the various orders of battle (air, ground, missile) that are real products of the Combat Intelligence Division (CID) within the Tactical Air Control Center (TACC) (TACP 50-29, 1984).

3.1.2.2 PLANNER. A principal product of planning within the TACC is the air tasking order (ATO), which is the responsibility of the Combat Operations Division (COD) (TACP 50-29, 1984). The functional capability of PLANNER is to make a simple assignment of a number of aircraft to the targets, in priority, until the supply of aircraft is exhausted. In this way, the ATO for the next day (or next planning period) will be formulated and made available to SIMULATOR for execution. During the assignment process, PLANNER will request and receive information about the number of aircraft at individual bases.

Input requirements for PLANNER include the prioritized targets and unit status. The output capability for PLANNER, given the above input, is the unit tasking.

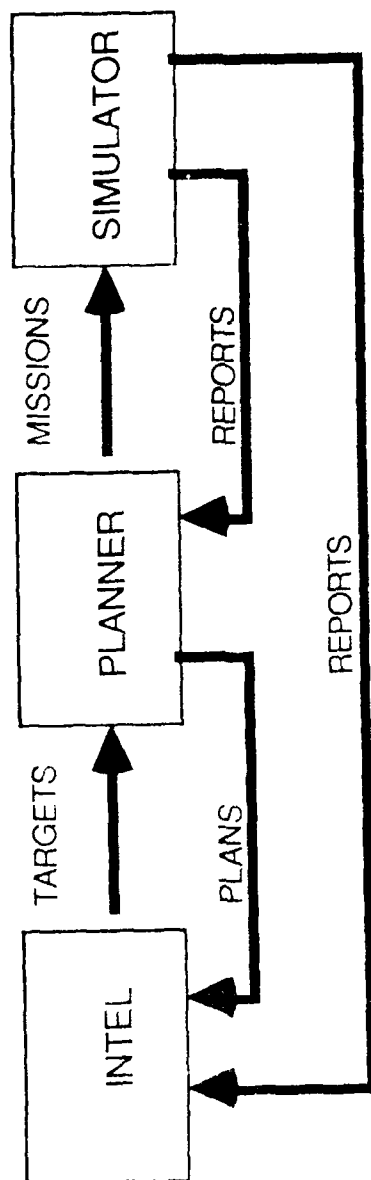


FIGURE 3-2
TAC-1 DOMAIN COMPONENTS

3.1.2.3 SIMULATOR. SIMULATOR is the system which emulates a knowledge-based simulation. SIMULATOR completes the C² process loop for the other two systems, thereby providing the environmental response and feedback on the changes to friendly and target status.

SIMULATOR first accesses the status of both the Red targets and the Blue units. The first function of SIMULATOR is to simulate a *reconnaissance of the Red targets* and report a list of sensed targets. Only targets sensed as operational are reported. These sensed targets are written out to the Common Database as the reported targets.

The second function of SIMULATOR can be activated after PLANNER has finished the unit tasking of the AIO. After accessing the unit tasking, SIMULATOR simulates flying of the missions against the targets. A Monte Carlo procedure is used to determine whether the aircraft or the target is destroyed. The unit status is changed to reflect aircraft losses. Subsequent reconnaissance produces a new list of sensed targets to be acted upon in the next cycle of activities in the TAC-1 demonstration.

The initial input requirements for SIMULATOR are the unit status and targets. Before simulating the missions, SIMULATOR must access the unit tasking.

The output capabilities of SIMULATOR are therefore the reports of sensed targets and unit status, both of which are stored in the Common Database for access by the other systems.

The simulation is time-stepped and uses a clock to control events. CLOCK is an ERIC actor common to SIMULATOR, PLANNER and INTEL. The clock in SIMULATOR is the master clock; each clock keeps a list of *actors-running*, which is a list of pairs of times and actors, indicating the time in the future at which the actor has a plan to do something. When the time advances past that event time, the actor is told to take the action that it has stored in its *schedule* list.

3.2 TAC-1 Demonstration

The normal sequence of steps in the demonstration is depicted in Figure 3-3 and described below.

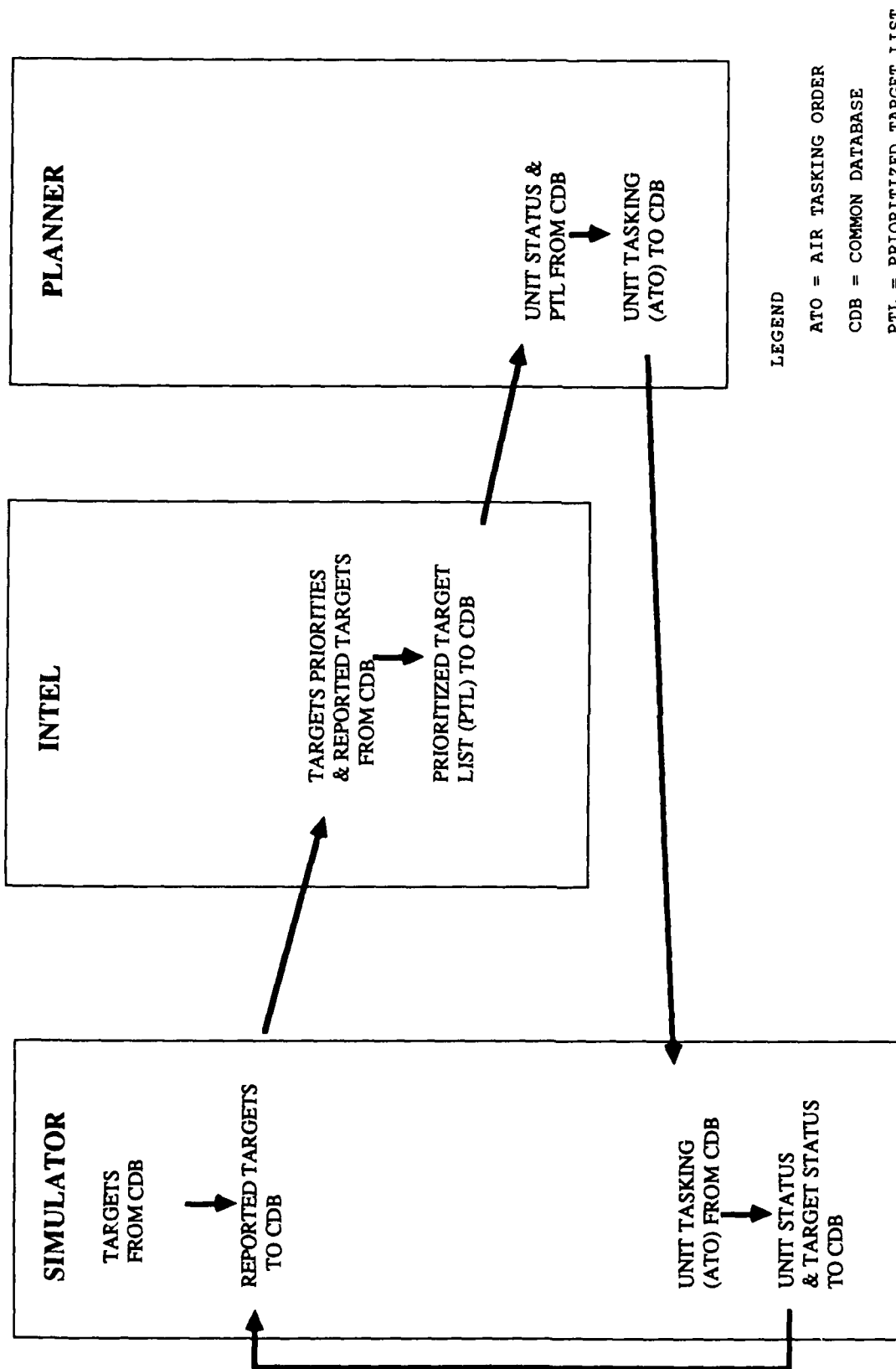


FIGURE 3-3
TAC-1 DEMONSTRATION SEQUENCE, FUNCTIONAL FLOW DIAGRAM

1. SIMULATOR is told to take the first step through the following message:

(ask SIMULATOR report sensed targets)

When SIMULATOR checks to see if targets are stored locally (on its property list), it finds it needs to get the data from the Common Database and does so by sending itself the following message:

(ask SIMULATOR initialize targets)

In the behavior for initialize targets, SIMULATOR sends its System Interface (SI) a message to get the target data from the database. The message is relayed through the SIMULATOR's Router Interface (RI) to the Router. The Router sends the database request through the interfaces for the Common Database wherein the appropriate data in the database relation target is accessed and returned through the same processes to SIMULATOR.

With the target data, SIMULATOR continues in its report sensed targets behavior by then simulating reconnaissance of the targets whereby some of the operational targets are reported as being sensed. SIMULATOR stores the reported targets in the Common Database's relation reported-target. This storage is accomplished through the appropriate interfaces and Router, similar to the data query above.

2. INTEL is then activated through the following message:

(ask INTEL prioritize targets)

whereupon it checks its property list to see if it has the current list of targets. As it does not, it will send itself the following message:

(ask INTEL get targets)

which initiates the service request to get-reported-targets. Router sends the request to the CKB because that is listed as one of CKB's capabilities. The CKB looks first in the CDB for the data and then, if it is not available there, initiates a service request which, through the interfaces and Router, results in the SIMULATOR being sent a message to report sensed targets. The result in both cases is for INTEL to have access to the data from the database relation reported-target as stored by SIMULATOR. After receiving the target data, INTEL then checks that its list of target priorities is current. As it is not, INTEL sends itself the following message:

(ask INTEL get target priorities)

which activates the database request to get the list of target priorities from the database relation *target-priority*. After receiving that data, INTEL continues with its behavior to prioritize targets. The targets are assigned priorities and the new relation *prioritized-target* is stored in the Common Database.

3. PLANNER is then activated through the following message:

(ask PLANNER produce ato)

for it to produce the air tasking order (ATO).

PLANNER checks its local data to see if it has the current data on the status of units. Finding that not to be the case, PLANNER sends itself the following message:

(ask PLANNER get unit status)

The message to *get unit status* activates the PLANNER behavior which sends a request to the database through the interfaces and the Router. The database relation *unit-status* is accessed for the appropriate data. Having received the unit status data, PLANNER checks that it has the prioritized target list (PTL) and, finding that not to be the case, sends itself the following message:

(ask PLANNER get prioritized targets)

The PLANNER behavior for this messages activates a service request to *get-prioritized-targets* which is directed by the Router to the CKB. The CKB first searches the CDB and then, if the data are not available and current, sends a service request which results in INTEL being sent a message to *prioritize targets*. In either case, the result is that PLANNER has access to the *prioritized-target* relation.

PLANNER assigns aircraft sorties from the appropriate units against the targets in order of priority. This information is stored in the Common Database in the relation *unit tasking*.

4. SIMULATOR is then reactivated through the following message:

(ask SIMULATOR fly missions)

in order for the ATO to be executed. SIMULATOR then simulates the receipt of the ATO by the Wing Operations Center (WOC), the consequent sorties being flown against targets, and the subsequent assessment of mission success or failure. To do this, SIMULATOR first issues a service request to **get-unit-taskings**, which goes to the CKB. The CKB either gets the data from the CDB or issues a service request to the PLANNER to **produce the ato**. In either case, the result is that SIMULATOR has access to the necessary information for it to obtain the necessary data to fly and assess the missions. The mission outcome is determined through a random draw against a probability of mission success. For a mission success, the **target** relation is updated to change the target status to non-operational. For a mission failure, the **unit-status** relation is updated to decrement the number of available aircraft.

5. The demonstration can continue with the message

(ask SIMULATOR report sensed targets)

The difference is that in checking for the current targets, the database query will ignore those targets which have been declared non-operational. Similarly, the new **unit-status** will reflect the results of lost aircraft on previous missions.

An alternative sequence of initiating action in the demonstration avoids actors in the KBS's having to send messages to themselves for some of the data, as follows:

(ask SIMULATOR initialize targets)
(ask SIMULATOR report sensed targets)
(ask INTEL get targets)
(ask INTEL get target priorities)
(ask INTEL prioritize targets)
(ask PLANNER get ptl)
(ask PLANNER get unit status)
(ask PLANNER produce ato)
(ask SIMULATOR get ato)
(ask SIMULATOR fly missions)
(ask SIMULATOR report sensed targets)

3.3 Prototyping Environment

The TAC-1 demonstration was developed while the concepts for the KB-BATMAN Shell continued to evolve. Hence, TAC-1 has required the use of rapid software prototyping techniques.

An object-oriented programming style was adopted for use in programming both the Shell and the TAC-1 domain systems. Object-oriented programming provides an abstraction in which real-world entities can be modeled as objects with behaviors and states (Stefik and Bobrow, 1986). Object-oriented programming proved to be beneficial for implementing rapidly changing software to match design changes during development. Window-based graphics was considered important for building a user interface to allow inspection and monitoring of interactions among TAC-1 components. A user interface is useful not only during demonstrations, but also during software development as a debugging tool since it can be difficult to follow the activities of concurrently executing systems.

TAC-1 software was developed incrementally using the Symbolics LISP machine's software engineering environment, which is recognized as one of the best for rapid prototyping (Weiss, 1986; Briggs, Jr., undated). TAC-1 operates under the Genera 7 version of the Symbolics operating environment. The general-purpose KB-BATMAN Shell software was written in Symbolics COMMON LISP and Flavors; Flavors is an object-oriented programming language embedded in LISP (Moon, 1986). The Air Force tactical domain software components were written primarily in ERIC (Hilton, 1987); ERIC is an object-oriented simulation language embedded in LISP which is based on ROSS (McArthur, Klahr, and Narain, 1984).

TAC-1 can be executed in a configuration of one or more LISP machines, with communications between machines handled by Chaos network software. During TAC-1 development, it was convenient to run all hosted systems on one machine; for demonstrations, one or more systems can be associated with each machine. Each hosted system represents a module which can be plugged into TAC-1 from any LISP machine which can communicate with a subnet machine which performs message routing.

While TAC-1 presently operates only on Symbolics LISP machines, the techniques and conventions used to construct hosted systems should be applicable to other hardware and software configurations.

3.4 Description of the KB-BATMAN Shell

Our testbed design envisions a number of distinct, cooperating knowledge-based systems. Each system may be hosted on a separate computer. Figure 3-1 depicts the logical connectivity of TAC-1 systems in a star network. The center of this star network is a *subnet*. Management of communications between systems is performed within this subnet, which operates on a single computer in the current implementation. The subnet consists of a Router and Router Interface (RI) objects for each system. The subnet also is a star network: each interface's communications pass through the Router on the subnet machine. Private communications between two systems is not encouraged by the TAC-1 design. Private communications increase the level of interdependence of participating systems, which is contrary to the requirement that component systems be loosely coupled.

Each hosted system is represented by a System Interface (SI) which links the subnet with the system's capabilities.

The following subsections describe the currently implemented functionality of the KB-BATMAN Shell. Refer to Appendix A for the software aspects of the implementation, including examples with LISP code.

3.4.1 Processes

To support asynchronous execution of requests and to reduce the likelihood of communications bottlenecks, multiple *processes* are involved in the KB-BATMAN Shell scheme. A process is a software entity which emulates a single hardware processor; conceptually each process is viewed as executing a separate computer program independently and concurrently. When multiple processes must execute on a single hardware processor, a control program called a scheduler is used to enable each process to execute for a short time. This technique is called time-slicing. The following types of processes are used in the testbed and are described in detail below:

- The Router (one process)
- Testbed Connection Monitor (one process)
- Router Interfaces (RIs) (one process per hosted system)

- System Interfaces (SIs) (one process per hosted system)
- System Executors (one process per active request execution)

The term component has been used in this document to refer to a modular part of TAC-1. A component consists of one or more processes, and these processes can execute on one or more machines. The machine on which a process executes depends on its role in TAC-1. Each domain KBS in TAC-1 is a component, and parts of the KB-BATMAN Shell are components, namely the Router, Common Database (a system), and the Common Knowledge Base (a system). Figure 3-4 diagrams the connectivity among processes, and the following table correlates components with processes and machines.

<i>Component</i>	<i>Processes</i>	<i>Machine(s)</i>
Router	Router	Subnet
	Testbed Connection Monitor	Subnet
Any system	system's Router Interface	Subnet
	system's System Interface	Any
Process Pool	System Executors	Machines with SIs

For the purposes of our testbed design, it does not matter whether these processes are executing on a multitasking single processor machine, a parallel processing machine, or a collection of either or both types of machines. The current implementation assumes that all processes operating on a particular machine have equal access to shared memory, a requirement that may not be generally required.

3.4.1.1 Router. The Router supports centralized control of communications among hosted systems.

Each Router Interface declares to the Router the *services* its system can perform upon request. It is possible that a service can be performed by more than one system. When a system requests a service, the Router determines which system is

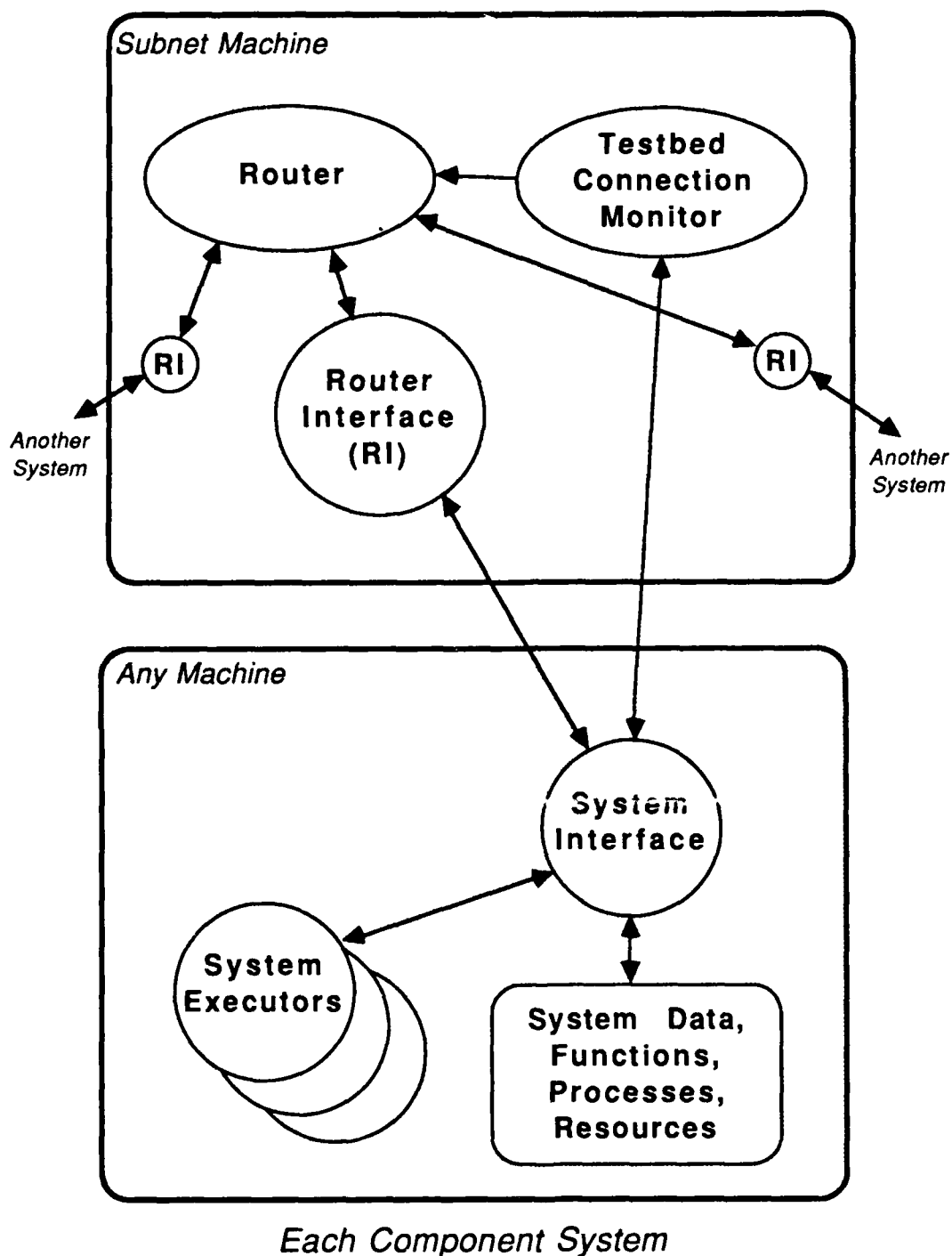


FIGURE 3-4
PROCESS CONNECTIVITY IN A TESTBED

most appropriate to perform the service by selecting one from the set of all declared servers. (In the present implementation no criteria are applied for selecting from multiple servers; one is chosen arbitrarily. Further work is required to incorporate selection criteria knowledge into TAC-1.) The requester does not address its request to any system; in fact, the requester does not know what system, if any, will perform a service. It is possible that a service cannot be performed, in which case the Router sends a error indication as a reply to the requester. The Router never interprets the contents of messages.

3.4.1.2 The Testbed Connection Monitor. The Testbed Connection Monitor (TCM) is a process operating on the subnet machine, the same machine as used by the Router. Its purpose is to handle requests from systems on other machines to connect themselves into the testbed. A system's System Interface initiates the request by sending a message to the subnet machine. TCM monitors the Chaos network for such requests-to-connect; when one arrives, TCM asks the Router object to create a Router Interface object for the system. The RI and SI then will open the necessary message streams to communicate with each other.

3.4.1.3 Interface Objects. A Router Interface and System Interface together form the communications interface between a system and the Router. The interface consists of two parts because the system may execute on a different machine than the Router, and there is a need to have processes on both machines poll for message arrival from either side, either the system or the Router. A system's RI process executes on the same machine as the Router; the SI process can operate on any machine, but typically is associated with a machine representing the domain system. (The processing within a system might itself operate on multiple machines.)

The externally-accessible capabilities of a system are defined as Flavor methods (behaviors) of an SI. A method specifies how to perform a service in the context of a system's operations. When an SI receives a service request from another system, it executes that request by evaluating the method asynchronously in a System Executor Process. In other words, the service request can be in execution while the SI continues to poll for further messages; in fact, multiple service requests can be in execution, each in a separate System Executor process. An SI also can receive a request from its own system, including from one of the System Executor processes it triggered. An SI only reads requests and notifications from its input port (described below); it does not read replies, whereas System Executors do.

3.4.1.4 System Executor Processes. A System Executor process is an independent process created on the same machine as a system's SI. When idle, System Executors are generic processes that can be reused by any SI operating on the same machine. The feature of reusability is an efficiency consideration since repeated creation and removal of processes is less efficient than reusing them on a Symbolics LISP machine.

A System Executor is asked by an SI to apply the SI's method function to specified arguments in order to execute a service request originating in another system. The System Executor can be considered to be performing the service in the context of the SI's system since it has access to any functions and data of that system. It is up to the method function to access those functions and data, and it may make service requests of other systems (or even its own system). If the request deserves a reply, the method function must explicitly send a reply via the SI. When a System Executor completes the method function application, it becomes idle until reused.

3.4.2 Input Ports

In our testbed design, when a message is transmitted, it is stored in a process' input port. In the current implementation, an input port can store an unlimited number of messages of arbitrary size. This lack of a limit certainly would not be acceptable in an operational environment, but is not considered important for the testbed at this time.

Each process has one input port, and the process may read messages from the input port in any order it chooses. Presently, most processes read and process all messages in a first-in, first-out (FIFO) manner. However, a System Executor process scans the contents of the input port of its currently associated System Interface for whichever reply messages it is expecting. A System Executor process does not process any request messages, and the System Interface process does not process any reply messages. A process which monitors a network connection for input will copy network input messages into its input port for subsequent processing.

3.4.3 Message Passing

There are several types of messages which can be transmitted in the testbed: requests, replies, and notifications.

A *request* is like a remote procedure call (RPC) (Liskov, 1982); a system states the desire to have a named service performed by an external handler, and expects a reply. A *reply* contains data in response to a specific request. A *notification* message is like an announcement and is not in response to a request and does not require a reply. Normally a message is delivered to the most appropriate system. However, a sending system may specify selectively that requests and notifications be *broadcast* to all components in the testbed, e.g., a clock time update.

A request is handled in the following way:

1. A system originates a request.
2. The system conveys the request to its SI on the same machine as the system.
3. The system interface transmits the request to a corresponding RI on the subnet machine, using Chaos network transmission if the system is on a different machine than the subnet. (Even if the system is on the same machine as the subnet, Chaos transmission may be used, optionally.)
4. The RI relays the request to the Router.
5. The Router looks in its internal database for available servers to handle the request. All servers' qualifications are compared to the preference criteria accompanying the message. If only one server meets the qualifications, the Router directs the request to that server's RI. If no servers meet the qualifications, the Router replies to the requester with an error indication. If several servers are applicable, the Router selects one.
6. The server's RI transmits the request to the server's SI, using Chaos network transmission if on a different machine, as in step 3.
7. The server's SI creates (or reuses) a separate process, called a System Executor, to execute the request asynchronously. The SI can continue to monitor for the arrival of new requests.

8. The System Executor invokes the requested behavior of the server system's SI. During execution of the request, the System Executor may make its own requests. When finished, the System Executor must explicitly send a reply to the request.
9. The System Executor instructs the server's SI to return the reply.
10. The SI transmits the reply to the RI.
11. The RI relays the reply to the Router.
12. The Router directs the reply to the RI for the requesting system as indicated in the reply.
13. The requester's RI transmits the reply to its system interface.
14. The requesting process in the system receives the reply. If the request was made synchronously, the requesting process waited for a reply matching the reply-id (system name and unique number) to arrive in the SI's input port. Waiting is performed by periodic polling. If the request was made asynchronously, the requesting process can check for a reply at any time.

A request contains the following fields of information:

- Tag indicating a request, :REQUEST.
- Name of requesting system, such as "INTEL".
- Unique request number for the requesting system.
- Request name, such as "GET-TARGETS".
- Any parameters to the request.
- Preference criteria for handling the request, in case there are several handlers.

Preference criteria are not supported in the current implementation; however, software hooks for their inclusion are present. They are intended to provide a way to choose a server based on the qualifications that the server has declared for a service. Qualifications may include level of authority, quality of results, speed of handling the service, and cost of handling the service.

A reply contains the following fields of information:

- Tag indicating a reply, :REPLY.
- Name of requesting system, such as "INTEL".
- Unique request number from the requesting system.
- Tag indicating whether the reply result is a value (:VALUE) that can be used directly by the receiver of the reply, or whether it is a form (:FORM) which must be evaluated locally in order to be meaningful to the receiver. This tag is used transparently by software supporting remote database access, since a database object cannot be transmitted from one machine to another; only a representation of the object can be transmitted and then reconstructed by the receiver into a local database object.
- The value or form which is the reply to a request.

A notification is similar in message form to a request, except that the unique message number for the requesting system is optional since no reply is anticipated by the sender. The Router determines which system(s) should receive the notification. A domain system should be able to enable or disable interest in being notified of any event, although this feature has not been implemented.

An example of using a notification in TAC-1 is when the Common Database system needs to inform systems when a database relation of interest to them is updated. A system expresses its interest in being notified by declaring the interest to the subnet, which is relayed to CDB. If notifications were not supported by CDB, each system would need to poll the database periodically to check for changes. The responsibility for producing database update notifications rests with the Common Database system.

Broadcasts are notifications or requests that are sent to all systems known to the Router, without regard for whether the system declared its support or interest for the particular message type. Broadcasts are used for very common functions which all systems can be expected to handle, such as a clock update notification, a request to report operating status, or a request to terminate. These messages are implicitly required services or notification interests inherent in all TAC-1 systems.

4.0 ASSESSMENT OF FIRST YEAR ACCOMPLISHMENTS

4.1 Summary of Accomplishments

In the first year of effort, the knowledge-based battle management testbed project has accomplished the following:

- Development of the KB-BATMAN Shell, a framework for integrating cooperating knowledge-based systems.
- Establishment of conventions and protocols for communications.
- Incorporation of a Common Database.
- Creation of a demonstration example as a proof-of-concept vehicle.

4.2 Assessment of TAC-1 Implementation

The ideas embodied in the current version of TAC-1 ignore a number of issues which need to be addressed in future work. Also, the implementation contains certain limitations which should be removed or better defined. Table 4-1 summarizes the assessment of TAC-1 in terms of the design issues discussed in Section 2.

DOMAIN SYSTEMS

The current TAC-1 demonstration consists of a simple scenario sufficient to demonstrate most of the features available in the KB-BATMAN Shell. However, a collection of systems with more substantial capabilities in the Air Force tactical domain would demonstrate the framework better and provide a more thorough test of TAC-1 concepts. Potential candidates for inclusion in TAC-2 (the successor to TAC-1) are two systems currently under development: a simulator at RADC and MITRE-Bedford's A Meta-Planning System (AMPS) (Tachmindji and Lafferty, 1986). Both of these systems use the CAMPS relational DBMS (developed at MITRE-Bedford) which is used in TAC-1. An additional system representing an intelligence node would be needed to coordinate with this simulator and planner. This intelligence node could be adapted from previous MITRE work on ANALYST, a tactical intelligence support system (Antonisse, Bonasso, and Laskowski, 1985) and OB1KB.

<i>Issue Area</i>	<i>First Year Status</i>	<i>Possible Improvements</i>
Centralization of Control	Centralized control.	Hybrid of centralized and decentralized.
Component Connectivity	Star network.	Cluster of star networks with gateways.
Common Functionality	Router.	Add knowledge for selecting from multiple servers.
	Common Database.	Combine with CKB into an object-oriented database.
	Common Knowledge Base -Small example only.	Build a substantial example incorporating Air Force doctrine.
Common Language	No systems need to use translations.	Include a system requiring translations.
Synchronization of Processing	Both synchronous and asynchronous processing are supported. Demonstration is entirely synchronous.	Emphasize asynchrony in domain systems.
Heterogeneous Hardware and Software	All systems use Symbolics LISP machines.	Include a system built for a different operating environment.
KBS Decomposition	Coarse-grained.	Possibly multigrained.
System Interdependency	Communications are service-based.	Include multiple servers for same service. Add preference criteria for service selection.
Robustness	Not addressed.	Demonstrate with examples where expected services are not performed.
Modular Behavior	Modularity is supported by KB-BATMAN Shell.	Demonstrate plugging systems in and out that support the same services.

TABLE 4-1
SUMMARY OF ASSESSMENT

a KBS for Army order of battle intelligence analysis (Schwamb and Kasif, 1986). TAC-1 concepts can also be applied to domains other than Air Force tactical battle management, such as strategic C² or tactical C² for other military services.

COMMON KNOWLEDGE BASE

Further work is necessary to provide centralized Air Force tactical domain knowledge in the Common Knowledge Base. This work must address issues such as what knowledge should be centralized in CKB, and what should remain in remote, specific-purpose domain systems.

HOMOGENEOUS IMPLEMENTATION

At present, all systems are implemented on Symbolics LISP machines and there is no need to translate between any system's internal language and the common language of the subnet. The structure of this common language needs to be defined sufficiently for new systems to be constructed on different machines. This common language undoubtedly will be defined in coordination with the definition of the Common Database and the Common Knowledge Base. The TAC-2 demonstration should include a system which needs to perform translations.

It may be useful to incorporate into TAC-1 a system implemented on different hardware (e.g., VAX/VMS) and perhaps using a programming language other than LISP (e.g., Ada or C). In doing so, additional issues may arise that were not evident when using LISP on a shared-memory LISP machine. A further test of the viability of the Router and subnet concepts would be to implement them on a different machine.

CONNECTIVITY

Use of a large number of systems suggests clustering systems into groups, each with a subnet and Router, and connecting these groups together so the entire set of systems can work in cooperation. This area for potential future work addresses the problems of centralized versus decentralized control. Flexibility of connectivity and reconfiguration should be a consideration in the development of any decentralized control mechanism.

ROBUSTNESS

Techniques for implementing robust behavior in systems must be developed and demonstrated. Robustness includes the ability for a system to deal with problems

associated with asynchronous communications, such as the failure to receive a timely response to a request and the potential for deadlocks and communications premised on outdated information.

MISCELLANEOUS

The TAC-1 software was designed for flexibility during incremental software prototyping. Issues concerning efficient operation of the Router and efficient use of resources such as the Common Database need to be addressed.

Support for dealing with multiple service handlers needs to be incorporated into the KB-BATMAN Shell. Examples of different quality service handlers should be programmed into component systems of TAC-2.

Currently, every request has a reply (like a remote procedure call). Future work should explore the issues involved with using notifications rather than requests to encourage less synchronous dependence among systems.

5.0 FUTURE DIRECTIONS

Future work on the Testbed should focus on further development of concepts embodied in TAC-1, including expansion of capabilities which were implemented with skeletal functionality. In addition, there are many aspects of cooperating distributed knowledge-based systems which remain to be addressed. The following recommendations are suggested for TAC-2:

- Use more substantive KBS systems in TAC-2.
- Formulate requirements for including knowledge in the Common Knowledge Base. Build a CKB appropriate for the component KBS systems in TAC-2.
- Use heterogeneous hardware, software, and information representations. Implement one system using hardware and software other than a Lisp machine and Lisp. Exercise protocol translations between heterogeneous systems.
- Demonstrate methods for implementing robust behavior in systems.

Further research is required in the following topic areas:

- Use of a Common Object Base which combines the features of the Common Database and Common Knowledge Base, and definition of protocols for knowledge abstractions and representations.
- Support for different types of system connectivities. In particular, it should be possible for sets of systems to be clustered together with localized central control, with gateway connections to other clusters of systems. Further study may be warranted to determine the differences and similarities between ABE's modules and TAC-1's systems, and how connectivity among components is handled in each approach. It would be worthwhile to determine what novel features from ABE might be beneficial to TAC-2, and vice versa.
- Feasibility of providing support for distributed problem solving in which systems may be fine-grained experts rather than broad experts, with greater levels of cooperation and communications than envisioned for TAC-1.

APPENDIX A

SOFTWARE DESCRIPTION

A.1 Overview

This appendix describes the software implementation of the KB-BATMAN Shell. Only the most prominent software functions of the Shell are documented; this appendix is not intended to be a complete user guide. Code excerpts from the TAC-1 component systems software are used in examples to demonstrate the use of the Shell. Note that work continues on the KB-BATMAN Shell, and therefore some details may change.

Testbed software was written in Symbolics COMMON LISP for operation with the Genera 7 version of Symbolics system software. In this appendix, the beginning of a description of a function, method (generic function), or variable is marked with a "»". LISP code is presented using the $\text{\texttt{TeX}}$ typewriter type style, e.g., `router`; examples use a smaller size, e.g., `router`. Function arguments are portrayed with a slanted style, e.g., *argument*.

A.2 Packages

All Testbed functions and variables reside in the `Testbed` package, which can be abbreviated as `TB`. All symbols intended to be used by other programs are exported from the `Testbed` package. It is highly recommended that programs be written in Symbolics COMMON LISP rather than ZETALISP.

Functions and symbols of the CAMPS relational database management system reside in the `DB` package. It is important to note that all attributes and symbolic values to be stored in Common Database should be in the `DB` package; otherwise name mismatches may occur between database accessors operating with different packages. This recommendation is due to the representation of symbolic and string data as symbols in the RDBMS. Further work may be warranted to remove this complication.

Unless otherwise noted, all references to LISP code in this appendix assume that the current package is `Testbed`.

A.3 Flavors and Methods

Generic Flavors is the object-oriented programming language used to implement the classes of objects participating in the `Testbed` along with their behaviors (methods). The following flavors are the most notable (sections given in parentheses describe the purpose of the objects):

- `router` (Section 3.4.1.1).
- `router-interface` (Section 3.4.1.3).
- `system-interface` (Section 3.4.1.3).
- `system-executor-process` (Section 3.4.1.4).
- `input-port-mixin` (Section 3.4.2).
- `interface-stream-mixin`. An interface stream supports communications between different machines.
- `db-si`, the system interface for the Common Database (Section 3.1.1.1).
- `kb-core-si`, the system interface for the Common Knowledge Base (Section 3.2.1.2).
- `testbed-frame`, the Monitor screen (Section A.9).

The built-in `si:process` flavor also is used.

Some flavors are intended to be instantiated only once, namely `router`, `db-si`, `kb-core-si`, and `testbed-frame`. Generic function methods are used to define behaviors for each flavor. For some methods, `:before` and `:after` methods are defined, mostly for attaching behaviors to methods to support the Monitor. In this appendix, the functions and methods necessary to initiate component processes and to enable them to communicate are described. Most of the functions that implement internal details are not described because their implementations may be changed.

A.4 Creating and Destroying Components

» (create-router)

Function

This function should be evaluated on the machine on which the Router is to execute as a process. If a Router already is executing, it is destroyed and a new one is created. A Testbed Connection Monitor process is created whose sole purpose is to monitor the Chaos network for attempts to connect to the Router. The user interface Monitor for viewing operations in the Testbed is created. The Router must be created before any other functions are executed which refer to the Router. The LISP object representing the Router is kept in the **router** variable, although programs outside the supplied Testbed software should not need to use it directly.

» (destroy-router)

Function

This function should be evaluated on the machine on which the Router is executing. The Router, any RIs, the Testbed Connection Monitor, and the user interface are all destroyed. Systems and SIs are not affected on any machine; however, any existing SIs cannot be used since they cannot communicate with a newly created Router in the current implementation.

» (reset-testbed)

Function

This function resets the Router to its initial state and destroys all RIs, SIs, and System Executors. It is identical to clicking on *Reset Testbed* in the Monitor control menu.

» (terminate-testbed)

Function

This function destroys the Router, all RIs, SIs, System Executors, and the Monitor. It is identical to clicking on *Terminate Testbed* in the Monitor control menu.

» (connect-si-to-subnet *system-name* *si-flavor-name* *router-machine*

*&key (force-use-of-chaos *force-use-of-chaos*)*)

Function

This function is used to create an SI and to connect it to an RI on the Router machine. The RI is created automatically on the Router machine via the Testbed Connection Monitor. The RI and SI normally will open network character streams for communicating with each other.

system-name is a string containing the system's name (e.g., "PLANNER"). By conven-

tion this name is in uppercase, although programs do not and should not distinguish case.

si-flavor-name is the name of the system interface's flavor, which should include **system-interface** as a component flavor mixin.

router-machine is a string or object which identifies the machine on which the Router is operating. If it is on the same machine as the system, use "Local". A Chaos communications link is established when the system and the Router are on different machines. When they are on the same machine, Chaos is not used unless **:force-use-of-chaos t** is used.

The value returned is the LISP object for the SI created.

Example from the INTEL system in TAC 1:

```
(defvar *intel-si*)
```

```
(defun setup-intel-si (&optional (machine "LOCAL"))
```

```
  "Set up the system interface for INTEL; the subnet is on the given machine."
```

```
  (setq *intel-si* (tb connect-si-to-subnet "INTEL" 'intel-si machine)))
```

```
→ (disconnect-si-from-subnet system-interface)
```

Method

This function is used to disconnect an SI from the subnet. The specified SI is destroyed, and the associated RI on the Router machine is destroyed.

```
→ (destroy-all-si)
```

Function

This function can be used to destroy all SIs on the current machine. It does not affect other processes associated with any systems.

A.5 Declarations

In order for the Router to know how to direct service requests and other messages, each system that has connected with the subnet must declare what services it claims to support and what notifications it is interested in receiving. The function

`declare-ri` is used for this purpose. In the current implementation, these declarations are made on the subnet machine; in a future version, they most likely will originate on the system's machine. `declare-ri` is intended to be used for initial declarations; more functions will be provided to permit additions or deletions of declarations to the Router.

`(declare-ri &key system flavor provides services interests)` *Function*

The `:system` keyword is required; otherwise this function has no effect.

`:system` is a string describing the system. Uppercase is suggested, although no programs should distinguish case.

`:flavor` is the LISP flavor from which the Router Interface for the system is instantiated. If omitted, the generic flavor `router-interface` is used.

`:provides` identifies the data messages produced by the system not directly in response to a request. These need not be declared, but the inclusion helps document how all involved systems cooperate. These should be given as strings in a list.

`:services` identifies the functions performed by the system in response to a request. This is necessary for a service request to be forwarded to the system by the Router. *Note that the system may always provide a service without being asked.* Services should be given as strings in a list.

`:interests` identifies the message/data items for which the system is interested in being notified whenever they are publicly issued/produced. This declaration essentially puts daemons on messages and data relations. Interests should be given as strings in a list.

Examples from all TAC-1 systems:

```
:: The Common Database system
(declare-ri :system "CDB"
            :services '("db-function"
                        "update-notification"))

:: The Common Knowledge Base system
(declare-ri :system "CEB"
            :services '("get-reported-targets"
```

```

        "get-prioritized-targets"
        "get-targets"
        "get-unit-status"
        "get-unit-taskings"))

(declare-ri :system "SIMULATOR"
  :services '("report-sensed-targets"
    "fly-missions"
    "time")
  :provides '("time"))

(declare-ri :system "INTEL"
  :services '("prioritize-targets"))
  :interests '("time"))

(declare-ri :system "PLANNER"
  :services '("produce-ato")
  :interests '("time"))

```

A.6 Defining Interfaces Between Systems and the Router

Interfaces between a system and the Router are defined using Flavors. Behaviors are defined as generic functions. Each system requires the definition of an SI object. Normally only one instance of a system and its interfaces are present in the Testbed. All of TAC-1's systems use a generic `router-interface`; any new RI flavor defined should include the provided flavor `router-interface`. An RI flavor optionally may be defined for each system if the generic `router-interface` flavor's methods are not sufficient for a particular system. Each SI flavor should include the flavor `system-interface`.

Example from the INTEL system of TAC-1:

```

;; -*- Package: User -*-
(defflavor intel-si ()
  (tb:system-interface))

```



```
(defmethod (prioritize-targets intel-si)
  (reply-id ignore)
  (tb:reply-from-system self reply-id
    (ask intel prioritize targets)))
```

It is suggested, although not mandatory, that the generic function syntax for Symbolics flavor method definition and invocation be used rather than the message-passing syntax (Moon, 1986). In the generic function syntax, a method name is not in LISP's keyword package, and is invoked just like any other function, e.g., (intel-method intel-object arg-1 arg-2). With message-passing syntax, a method name is a keyword, and is invoked by sending an object a message, e.g., (send intel-object :intel-method arg-1 arg-2). The KB-BATMAN Shell software uses the generic function syntax, but supports both styles for user-defined SIs.

```
>> (request-service-with-wait system-interface service
    &optional (arguments nil) (timeout nil)) Method
```

This function provides a synchronous way for a system to dispatch a query to its *system-interface* which will relay it to the Router, and then wait for a reply.

service is a symbol identifying the service requested. *arguments* is a list of the parameters for the service; use nil if there are none. *timeout*, if given, specifies how much time, is allowed to elapse waiting for a reply before giving up; it can be either sixtieths of a second, or nil for no timeout. The value returned is the response from the service handler, or :timeout if no response was received.

Example from the INTEL system of TAC-1:

```
;; -*- Package: User -*-
;; ERIC behavior definition for the Air-Force-Intel class of objects.

(ask air-force-intel when receiving (prioritize targets)
  (let (tgt-id tgt-type tuple-alist
        (view (tb:request-service-with-wait *intel-si*
          "get-reported-targets"))))
    (ask myself store old reported-targets)

    (loop with cursor = (db:new-cursor view)
```

```

until (null cursor)
do (setq tuple-alist (db:tuple-alist-at-cursor cursor))
(setq tgt-id
  (cdr (assoc 'db:target-report-id tuple-alist)))
(setq tgt-type
  (cdr (assoc 'db:target-type tuple-alist)))
(format t "~%Target id: ~A, target type: ~A"
  tgt-id tgt-type)
(ask myself add (list tgt-id tgt-type)
  to your list of reported-targets)
(setq cursor (db:forward-cursor cursor))))

```

» (request-from-system system-interface id service arguments) *Method*

This function provides an asynchronous way for a system to dispatch a service-request to its *system-interface*, which will relay it to the Router. *id* is a unique message identifier to be provided if a reply is solicited; the form (create-reply-id *system-interface*) can be used to create one, else use nil if no reply is desired. *service* is the name of the service to be requested, and *arguments* are any parameters to the service request, or nil if none are necessary.

Note that there is no assurance that any reply will be directed to the requester; on the other hand, a service request may result in any number of responses. Similarly, a reply may be either a direct response to the request, or a response to a number of requests from the requester and other systems. There may be no way for the responder, interfaces, or the Router to be able to associate a particular reply with a request. The unique *id* should be returned by a responder to assist a requester in correlating the response to a particular request. The requester may also provide a notification interest name as part of the *arguments* to the request; the responder can issue a notification for that interest when appropriate.

» (message-from-system system-interface service arguments) *Method*

This function is identical to (request-from-system system-interface nil *service arguments*), and is intended to be used for sending messages without soliciting a direct reply.

» (reply-from-system system-interface reply-id reply-message)

&optional (form-or-value :value))

Method

This function is used within a method for a system's service to return a reply to the requester of the service. The `reply-id` argument is obtained from the service method. (When an SI receives a service request, it invokes the corresponding service method to perform the service. The SI invokes the service method with two arguments, `reply-id` and `arguments`.) The contents of the reply are given by `reply-message`, which may contain any transmittable LISP form.

Put simply, a transmittable LISP form cannot contain references to complex LISP objects (e.g., a database relation object, structures, arrays; anything that has a print name using `#<...>`). As discussed in Section 3.2.3, complex LISP objects such as database relations can be transmitted as reconstructor forms which must be evaluated upon receipt on another machine. In order to be transmittable, it must be possible to derive reconstructor forms from an object and to produce a facsimile of the object on another machine.

Example from the INTEL system of TAC-1:

```
(defmethod (prioritize-targets intel-si)
  (reply-id arguments)
  (ignore arguments) ; (No arguments are expected)
  (tb:reply-from-system self reply-id
    (ask intel prioritize targets)))
```

Example from the Common Knowledge Base system of TAC-1:

```
(defmethod (get-reported-targets kb-core-si)
  (reply-id ignore)
  (when (db:empty-relation? "reported-target")
    ;; "Translate" the request into Report-Sensed-Targets
    (request-service-with-wait self "report-sensed-targets"))
  (reply-from-system self reply-id
    (producing-form
      (db:full-select "reported-target"))
    :form))
```

In the above example, `producing-form` is a macro which indicates to the RDBMS that a reconstructor form is desired as a result rather than a value, since the value

of `full-select` is a relation object which cannot be transmitted as a value. The opposite effect can be achieved by using `producing-value`, although producing a value is the conventional behavior of most services. (The use of the database function actually is a transparent usage of a service request, as described in Section A.7.2.)

» `(create-reply-id system-interface)` *Method*

This function creates a unique identifier associated with the `system-interface`, which can be used as an argument to `request-from-system`. This `reply-id` is transmitted with the service request and can be used by the Router to direct any replies back to `system-interface`.

» `(check-for-reply-message system-interface reply-id)` *Method*

This function is used to scan `system-interface`'s input port for a reply message tagged with the given `reply-id`. It returns `t` if one is found, or `nil` if none are present.

» `(read-reply-message system-interface reply-id)` *Method*

This function will read a reply message from `system-interface`'s input port that is tagged with `reply-id`. If the type of the reply is `:form`, the reply message is processed (as for database relation reconstruction), and the result is returned. If the type of the reply is `:value`, the actual value is returned. If no matching message was present, `nil` is returned.

» `(wait-for-reply-message system-interface reply-id`
 `&optional (timeout nil))` *Method*

This function combines `check-for-reply-message` and `read-reply-message`. If no reply matching `reply-id` is available, the current process will poll periodically for one.

» `(execute-in-system system-interface reply-id form`
 `&key (wait nil))` *Method*

This function is called by a service method of a system interface in order to execute a LISP form in its system's context asynchronously. The evaluation of the LISP form does not take place in the caller's process; it occurs in a separate process intended solely for such evaluation. This method does not await a response or result of evaluation.

A System Executor process is obtained, either one from a list of unused SE processes or a newly-created one, and the SE process is reset to evaluate *system-executor-eval* (described below) on the arguments *system-interface*, *reply-id*, *form*, and *wait*.

» (*execute-in-system-with-wait system-interface form*) *Method*

This method combines *execute-in-system* and *wait-for-reply-message*.

» (*system-executor-eval system-interface reply-id system-form wait*) *Method*

This is the top level function used by a System Executor process to evaluate the LISP form *system-form* for a system; it is triggered by the system's system interface process. Use of this function is layered beneath several other functions; application software will not need to deal with its intricacies.

The SE's behavior is complicated by the possibility that the evaluation of the form might be for local system needs and may result in service requests being issued, or it might be for the execution of a service requested by another system. If the evaluation is to produce a result from this system to be returned to the subnet, *reply-id* should be non-nil. If the evaluation is for local needs (e.g., a remote database function call), *reply-id* should be nil, and a *reply-id* may be embedded as an argument within the *system-form*, which typically is an invocation of the function *request-from-system*.

If *reply-id* is not nil, the System Executor sends a reply to the *system-interface* which triggered it, *self*. The reply message is either the result of the evaluation of the *system-form* when *wait* is nil, or the reply obtained by waiting for a message with *reply-id* when *wait* and *reply-id* are both non-nil.

This function also binds the value of **testbed-si** to *system-interface* for use by any RDBMS functions that might be invoked as a result of evaluating *system-form*. This binding permits database functions to use the appropriate SI for communicating with the subnet.

A.7 Database Access

Table A-1 lists the available RDBMS functions. Functions in this table marked with “†” are extensions to the CAMPS RDBMS (Brown and Schafer, 1987) written for use in the Testbed and are described in the following section. CAMPS RDBMS functions are listed but not documented here. The reader is assumed to be familiar with the concepts and terminology used for relational databases; see Date (1987).

The functions marked with a “*” can be used to access either local relations or relations in the Common Database. Each function with a “*” has a corresponding “-LOCAL” function (e.g., `FULL-SELECT` and `FULL-SELECT-LOCAL`). The “-LOCAL” function never will attempt to access the Common Database on a remote machine, whereas the regular function will. “-LOCAL” functions should not be called directly in application software; they are intended to be used by the RDBMS only on the CDB machine. All RDBMS symbols are in the DB package, as are the names of all relations defined by the RDBMS.

A.7.1 New Database Access Functions

The following functions are extensions to the RDBMS provided by Brown and Schafer.

» <code>(db:full-select relation)</code>	<i>Function</i>
Selects all attributes and tuples from <i>relation</i> .	

» <code>(db:sorting-select relation conditions)</code>	<i>Function</i>
Selects all tuples from the given <i>relation</i> , creating a new relation for which repeated uses of the function <code>db:forward-cursor</code> are guaranteed to point at successive tuples in a sorting order specified by the <i>conditions</i> argument. This guarantee expires when any modification is made to the relation.	

For now, the *conditions* may contain only one attribute and attribute comparison predicate. The *conditions* argument is a list of (*attribute attribute-compare-predicate*), e.g., `'((db:attr ,# '<))`.

Selection Functions

- * (DB:GENERIC-SELECT relation conditions)
- * (DB:EQUALITY-MEMBER-SELECT relation attribute-value-alist)
- * (DB:EQUALITY-SELECT relation attribute-value-alist)
- * (DB:NON-EQUALITY-MEMBER-SELECT relation attribute-value-alist)
- * (DB:NON-EQUALITY-SELECT relation attribute-value-alist)
- * (DB:RANGE-SELECT relation attribute-range-alist)
- * (DB:PROJECT relation &rest attributes)
- *† (DB:FULL-SELECT relation)
- *† (DB:SORTING-SELECT relation conditions)

Combination Functions

- * (DB:JOIN relation-a relation-b)
- * (DB:JOIN-ON-ATTRIBUTES relation-a relation-b attribute-pair-list)

Transaction Functions (not exercised in TAC-1)

- (DB:COMMIT-TRANSACTION &optional (relation-list *mod-relations*) delete)
- (DB:BACKOUT-TRANSACTION relation-list)
- (DB:SET-UP-LOCK relation-list)
- (DB:RESET-LOCK relation-list)

Modification Functions

- * (DB:UPDATE relation attr-val-cond new-attr-val
&optional unchanged-vals add-ok)
- * (DB:INSERT-ATTRIBUTE relation attr-property-list)
- * (DB:ADD-TUPLE relation values-alist)
- *† (DB:ADD-TUPLES relation values-alist-list)
- *† (DB:DELETE-ALL-TUPLES relation)
- * (DB:EQUALITY-DELETE relation attribute-value-alist)

* *Support both local and remote access*

† *Extensions to CAMPS RDBMS*

TABLE A-1
RELATIONAL DATABASE FUNCTIONS

Value Retrieval Functions

- * (DB:SINGLETON? relation)
- * (DB:VALUES-FROM-RELATION relation attribute &optional max)
- * (DB:EMPTY-ATTRIBUTE? relation attribute)
- * (DB:EMPTY-RELATION? relation)

Control Flow Functions

- (DB:NEW-CURSOR relation)
- (DB:FORWARD-CURSOR cursor)
- (DB:VALUE-FROM-TUPLE-AT-CURSOR cursor attribute)
- (DB:VALUE-RAW-FROM-TUPLE-AT-CURSOR cursor attribute)
- † (DB:TUPLE-AT-CURSOR cursor)
- † (DB:TUPLE-ALIST-AT-CURSOR cursor)
- (DB:MODIFY-TUPLE-AT-CURSOR cursor attribute value)
- (DB:DELETE-TUPLE-AT-CURSOR cursor)

Data Dictionary Functions

- (DB:DEFREL props &rest tuples)
- * (DB:DELETE-RELATION relation)
- (DB:CREATE-RELATION &rest keylist)
- * (DB:CREATOR-FOR-RELATION relat)
- * (DB:SIZE-OF-RELATION relation)
- * (DB:ATTRIBUTES-OF-RELATION relation)
- * (DB:FILE-OF-RELATION relation)
- * (DB:DOCUMENTATION-OF-RELATION relation)
- * (DB:HISTORY-OF-RELATION relation)

* *Support both local and remote access*

† *Extensions to CAMPS RDBMS*

TABLE A-1
RELATIONAL DATABASE FUNCTIONS
(CONCLUDED)

» (db:tuple-at-cursor *cursor*)

Function

Gets the tuple at the current *cursor* position; all values are returned in external format in a list. Code should not rely on the order of values; the function *db:tuple-alist-at-cursor* is recommended instead of this one. Example:

```
Command: (setq cursor (db:new-cursor db:ac-caps))
#<cursor-struct for #<relat-struct AC-CAPS 397> 471>
Command: (db:tuple-at-cursor cursor)
(A-10 ALL-WEATHER-AC)
```

» (db:tuple-alist-at-cursor *cursor*)

Function

Gets the tuple at the current *cursor* position; all values are returned in external format in an association list keyed on attributes. Code should not rely on the order of associations in the list. Example:

```
Command: (db:tuple-alist-at-cursor cursor)
((DB:CAPABILITY DB:ALL-WEATHER-AC) (DB:AIRCRAFT-NAME DB:A-10))
```

» (db:add-tuples *relation values-alist-list*)

Function

Adds multiple tuples to *relation*. This is like the provided *db:add-tuple* except that the argument is a list of tuples rather than just one.

» (db:delete-all-tuples *relation*)

Function

Deletes all tuples from the given *relation*.

» (db:create-create-a-relation *relation*
 &optional (*new-name-of-relation* nil))

Function

Returns a LISP form which, when evaluated, will create a relation with contents identical to the specified *relation*. Normally a unique name for the relation will be generated automatically, and will start with *TEMP-DB-REL-*. However, a name for the relation can be specified as the optional argument *new-name-of-relation*.

This above function is used internally by the non-“-LOCAL” database functions and is transmitted to a remote machine in order to reconstruct a relation.

» (db:reload-relation relation)

Function

Given *relation*, reloads its definition from the file from which it last was loaded.

A.7.2 Remote Database Access

As described in Section 3.1.1.1, the conventional way to access a Common Database relation is to cite it by name, e.g., "REPORTED-TARGETS". Alternatively, a symbol can be used, e.g., 'DB:REPORTED-TARGETS. When the relation name (or symbol) is used as an argument to a database function, the function will be able to access the relation in the Common Database, on whatever machine that relation resides. For example, the following LISP form will obtain from the CDB a view of the relation "REPORTED-TARGETS" and return to the invoker a relation object containing that view:

```
(setq target-view (db:full-select "REPORTED-TARGETS"))
```

The relation object returned will be a local view; it never will be the actual relation object present in the CDB. Software may manipulate local views of relations in any way without affecting the CDB.

In order to use a database function to manipulate an arbitrary relation object rather than a relation present in the CDB, the object itself must be given as the relation argument to the function, as in the following example:

```
(setq live-targets  
  (db:equality-select target-view  
    '((db:status db:live))))
```

Cursor operations may be applied only to local views, never directly to a CDB relation.

In order to modify a CDB relation, one of the modification functions listed in Table A-1 must be applied to a relation identified by name.

To summarize RDBMS operations: relations in the Common Database are accessed by *name*; relation views are accessed by *reference*.

Each database function which accesses a relation by name transparently makes a service request to the subnet for the execution of a database service (called db-function). It also waits for a reply to the service request, interprets the reply (if a relation needs to be reconstructed), and returns the value of the reply to its caller. Hence, remote database users do not have an inflexible connection to a Common Database; a different CDB could be substituted at any time.

A.8 Common Knowledge Base

The current version of the CKB contains only a few behaviors and cannot be considered knowledge-based. The CKB is the component of TAC-1 which requires the greatest future efforts. The current CKB contains the following behaviors: **get-reported-targets**, **get-prioritized-targets**, **get-target-priorities**, **get-targets**, **get-unit-status**, and **get-unit-taskings**. Each of these behaviors supplies a requester with data from the CDB; if the requested data are not present, the behavior will make sure it is by issuing the appropriate service request to generate the data.

A.9 User Interface

Monitoring software can be active on any machine in which a testbed process is executing. Normally the Monitor operates on the subnet machine where the Router and Router Interfaces are executing. The Monitor, if active on a machine, will display message transmissions and process states for all testbed processes executing on that machine.

The Monitor screen consists of the following window panes:

- Router message activity pane.
- Message activity panes for up to four systems. Messages processed by a system's system interface and router interface will appear in the same pane.
- A single message activity pane for any other processes. These processes may represent systems beyond the four for which dedicated panes are allocated:

they also may be from external agents such as a human user's input typed to a LISP Interaction pane, and from System Executor processes.

- A LISP Interaction pane where a human user can enter any LISP form to be evaluated, including commands to any system active in the testbed.
- A Testbed Services pane where each active system is listed along with all services it has declared that it can provide.
- A Testbed Options pane in which the human user monitoring testbed activity can choose parameter values, including whether to monitor messages at all; how much time, if any, to pause between messages; and the number of lines to show from each message.
- A Command Menu pane containing commands to control views of different types of information, including help text for operating the Monitor; clearing all messages from the message panes; selecting different configurations for panes on the screen; presenting summary information for the Router and all interface processes including contents of input ports and current state of execution; and describing the instance variables for the Router or an interface process. Also present in this menu are commands for resetting the testbed, where all interface processes are destroyed, and for terminating the testbed, where all processes and screens are destroyed.
- A Control Menu pane containing commands for controlling execution of processes, including suspending all interface processes; resuming all interface processes; and stepping one message at a time for all processes with input ports.

Figure A-1 contains an example of the testbed Monitor screen.

The Monitor can be used as a debugging tool when creating systems and integrating them into TAC-1. Debugging asynchronous systems can be more difficult than debugging synchronous systems since there is a greater potential for obscure coordination and timing problems, deadlock of execution, and bottlenecks of messages.

Even in a fully-developed operational suite of systems, a monitoring capability would be a useful asset for each component system. A knowledge-based system's behavior may vary according to its perception of the responsiveness of its partners in the suite of systems.

Router (GENTEMP "TEMP-DB-REL-") [11:25:30] Router is sending a message to PLANNER-RI: (:REPLY ("PLANNER" . 4) (:FORM (DB:CREATE-A-RELATION :RELNAME ...	<input type="checkbox"/> TAC-1 Services <input checked="" type="checkbox"/> Router <input checked="" type="checkbox"/> INTEL PRIORITIZE-TARGETS <input checked="" type="checkbox"/> DB DB-FUNCTION UPDATE-NOTIFICATION <input checked="" type="checkbox"/> SIMULATOR REPORT-SENSED-TARGETS FLY-MISSIONS <input checked="" type="checkbox"/> PLANNER PRODUCE-ATO <input checked="" type="checkbox"/> KB-CORE GET-REPORTED-TARGETS GET-PRIORITIZED-TARGETS GET-UNIT-TASKINGS
DB (DB:CREATE-A-RELATION :RELNAME [11:25:29] DB-RI is sending a message to Router: (:REPLY ("PLANNER" . 4) (:FORM (DB:CREATE-A-RELATION :RELNAME ...	
KB-CORE (DB:CREATE-A-RELATION :RELNAME [11:24:49] KB-CORE-RI is sending a message to Router: (:REPLY ("PLANNER" . 3) (:FORM (DB:CREATE-A-RELATION :RELNAME ...	
SIMULATOR (:REQUEST ("SIMULATOR" . 24) TBED:GET-UNIT-TASKINGS 27) [11:23:30] SIMULATOR-RI is sending a message to Router: (:REQUEST ("SIMULATOR" . 24) TBED:GET-UNIT-TASKINGS 27)	<input type="checkbox"/> Options <input checked="" type="checkbox"/> Message Monitoring On Off <input checked="" type="checkbox"/> Message Pause None 1/2 sec 1 sec 2 sec 5 sec <input checked="" type="checkbox"/> Output Message Maximum Lines 1 2 5 10
INTEL [11:19:45] INTEL-SI is receiving a message from its RI: (:REPLY ("INTEL" . 1) (:FORM (DB:CREATE-A-RELATION :RELNAME [11:19:46] Reply for INTEL request # 1 was received.	
Other Message Sources [11:25:42] PLANNER-SI is receiving a message from its RI: (:REPLY ("PLANNER" . 4) (:FORM (DB:CREATE-A-RELATION :RELNAME ...	<input checked="" type="checkbox"/> Help Refresh Screen Clear Messages on Screen Configure Screen Summarize Components Describe Component Reset Testbed Terminate TAC-1
<input type="checkbox"/> Command: (ask simulator fly missions) [11:25:01 Process System Executor 2 wants to type out Select Background Dynamic Lisp Interactor 2 by typing Function-0-S.] <input type="checkbox"/> Lisp Interaction	<input type="checkbox"/> Suspend RI Processes <input type="checkbox"/> Resume RI Processes Step

Mon 31 Aug 11:25:55 Rick

CL-USER: Reply Wait

PERSEUS

FIGURE A-1
TAC-1 DEMONSTRATION: MONITOR SCREEN

A.10 Summary of Operation

This section briefly summarizes the steps required to operate the TAC-1 demonstration.

1. On each Symbolics LISP machine to be used, enter the command "Load System Testbed". This will load all KB-BATMAN Shell software.
2. Load hosted system software on desired machines.
3. On the machine to be the subnet machine with the Router, evaluate "(create-router)", or, alternatively, type the keystroke *Select Network*. This will create the Router process and the Testbed Connection Monitor process. It also will activate the Monitor user interface.
4. On the subnet machine, evaluate the declarations for each system (the **declare-ri** forms). Note that these declarations can be made independently of the operation of any of the hosted systems.
5. For each hosted system to be plugged into the Testbed, including CDB and CKB, evaluate a **connect-si-to-subnet** form. The value returned will be the SI object created for the system.

A system may start sending messages to the subnet once the system has connected to the subnet successfully.

REFERENCES

References Cited

Agha, Gul A. (1987). *Actors: A Model of Concurrent Computation in Distributed Systems*. Cambridge MA: MIT Press.

Antonisse, H. J., R. Peter Bonasso, and S. J. Laskowski (April 1985), *ANALYST II: A Knowledge-Based Intelligence Support System*, MTR-84W00220, McLean VA: The MITRE Corporation.

Benoit, John W. et al. (April 1986). *AirLand Loosely Integrated Expert Systems: The ALLIES Project*, MTR-86W00041, McLean VA: The MITRE Corporation.

Briggs, Jr., Richard S. (undated, 1985?), "Evaluation of a Symbolics 3640 as a Rapid Prototyping Environment", Milford OH: International TechneGroup Inc.

Brown, Richard H. and Alice L. Schafer (1 May 1987), *A Relational Database Management System for CAMPS*, Draft, Bedford MA: The MITRE Corporation.

Date, C. J. (1987). *A Guide to the SQL Standard*. Reading MA: Addison-Wesley.

Erman, Lee D., Jay S. Lark, and Frederick Hayes-Roth (May 1986), *Engineering Intelligent Systems: Progress Report on ABL*, TTR-ISE-86-102, Palo Alto CA: Teknowledge, Inc.

Gien, Michel and Hubert Zimmerman (1979). "Design Principles for Network Interconnection", *Proceedings, Sixth Data Communications Symposium*, November 1979, pp. 109-119; reprinted in Thurber, Kenneth J., ed. (1980), *Tutorial: A Pragmatic View of Distributed Processing Systems*, pp. 365-375, Long Beach CA: IEEE Computer Society.

Graham, Richard A. (March 1983), *An Environment for Distributed Simulation of Command and Control Networks*, Master of Science Thesis, Monterey CA: Naval Postgraduate School.

Green, P. E. (1979). "An introduction to network architectures and protocols", *IBM Systems Journal*, Volume 18, Number 2, pp. 202-222; reprinted in Thurber, Kenneth

J., ed. (1980), *Tutorial: A Pragmatic View of Distributed Processing Systems*, Long Beach CA: IEEE Computer Society.

Hilton, 1LT Michael L. (July 1987), *ERIC: An Object-Oriented Simulation Language*, RADC-TR-87-103, Griffiss Air Force Base NY: Rome Air Development Center.

Liskov, Barbara (May 1982), "On Linguistic Support for Distributed Programs", *IEEE Transactions on Software Engineering*, Volume SE-8, Number 3, pp. 203-210.

McArthur, David, Philip Klahr, and Sanjai Narain (December 1984), *ROSS: An Object-Oriented Language for Constructing Simulations*, R-3160-AF, Santa Monica CA: Rand.

Moon, David A. (November 1986), "Object-Oriented Programming with Flavors", OOPSLA-'86 Conference Proceedings, published as *ACM SIGPLAN Notices*, Volume 21, Number 11, pp. 1-8.

Peterson, Robert W. (March 1987), "Object-Oriented Data Bases", *AI Expert*, pp. 26-31.

Schwamb, Karl S. and Emily H. Kasif (December 1986), *OB1KB: A Knowledge-Based System for Army Order of Battle Intelligence Analysis*, MTR-86W00133, McLean VA: The MITRE Corporation.

Stefik, Mark and Daniel Bobrow (Winter 1986), "Object-Oriented Programming: Themes and Variations", *AI Magazine*, Volume 6, Number 4.

Tachmindji, Alexander J. and Edward L. Lafferty (June 1986), "Artificial Intelligence for Air Force Tactical Planning", *Signal*, pp. 110-114.

TACP 50-29, General Operating Procedures for Joint Attack of the Second Echelon (J-SAK), *USREDCOM Pam 525-8*, *TRADOC Pam 525-45* (31 December 1984), Langley Air Force Base VA: United States Air Force Tactical Air Command.

Thomas, Ivan. Ronald Foss, and Scott Hosking (5 November 1986), *Advanced Tactical Air Control Center, Functional Description. ITACC-Phase IV*, St. Paul MN: The Sperry Corporation.

Walter, Sharon M. (1 October 1986), *Technical Objectives and Plans; FY87; RADCS; Knowledge-Based Battle Management Testbed*, Rome, NY: Rome Air Development

Center.

Weber, Roger W. (March 1987), *Decision and Development Support Environment (DADSE) Program*, briefing slides, Griffiss Air Force Base NY: Rome Air Development Center/COAD.

Weiss, Andrea H. (November 1986), "An Order of Battle Advisor", *Signal*, pp. 91-95.

Yonezawa, Akinori and Mario Tokoro (1987), *Object-Oriented Concurrent Programming*, Cambridge MA: MIT Press.

Other References

Battle Staff Course Workbook (October 1985), Hurlburt Field FL: United States Air Force Air Ground Operations School.

Franta, William R., Helmut K. Berg, and William T. Wood (October 1982), "Issues and Approaches to Distributed Testbed Instrumentation", *IEEE Computer*, Volume 15, Number 10, pp. 71-81.

Hewitt, Carl (1986), "Concurrency in Intelligent Systems", *AI Expert*, Premier 1986, pp. 45-50.

Howard. H. Craig and Daniel R. Rehak (July 1987), "KADBASE: A Prototype Expert System-Database Interface for Integrated CAE Environments", *Proceedings, National Conference on Artificial Intelligence (AAAI-87)*, Volume 2, pp. 804-808.

Lane, Jr., LTC John J. (1981), *The Air War in Indochina, Vol I, Monograph I: Command and Control and Communications Structures in Southeast Asia*, Maxwell Air Force Base AL: Air University, Air War College.

Lesser, Victor R. and Daniel D. Corkill (Fall 1983), "The Distributed Vehicle Monitoring Testbed", *AI Magazine*, Volume 4, Number 3, pp. 15-33.

McArthur, Dave, Randy Steeb, and Stephanie Cammarata (1982), "A Framework for Distributed Problem Solving", *Proceedings, National Conference on Artificial Intelligence (AAAI-82)*, pp. 181-184.

Misra, Jayadev (March 1986), "Distributed Discrete-Event Simulation", *ACM Computing Surveys*, Volume 18, Number 1, pp. 39-65.

Shatz, Sol M. (June 1984), "Communication Mechanisms for Programming Distributed Systems", *IEEE Computer*, Volume 17, Number 6, pp. 21-28.

GLOSSARY

ABE	A module-oriented AI development environment tool under development at Teknowledge, Inc. (Erman, Lark, and Hayes-Roth, 1986).
Ada	A computer programming language developed and controlled by the United States Department of Defense, Ada Joint Program Office, for use in embedded military systems.
AGOS	Air/Ground Operating School
AI	Artificial Intelligence
ALLIES	AirLand Loosely Integrated Expert Systems, developed by MITRE-Washington (Benoit et al., 1986).
AMPS	A Meta-Planning System, under development at MITRE-Bedford.
ANSI	American National Standards Institute.
ATO	Air Tasking Order.
C	A computer programming language.
C ²	Command and control.
CAMPS	Core of A Meta-Planning System, under development by MITRE-Bedford (Brown and Schafer, 1987).
CDB	Common Database.
CID	Combat Intelligence Division.
CKB	Common Knowledge Base.
COD	Combat Operations Division.

COMMON LISP	A computer language often used for AI programming; it is in the process of ANSI-standardization.
CPD	Combat Plans Division.
DB	Database.
ERIC	An object-oriented simulation language based on ROSS and written in COMMON LISP (Hilton, 1987).
Flavor	The definition of a class of objects in Flavors.
Flavors	An object-oriented programming language which is an integral part of Symbolics' version of COMMON LISP (Moon, 1986).
Genera	The operating system for Symbolics LISP machines.
Hosted System	Same as System.
Input Port	A software entity in which messages are queued.
INTEL	The intelligence KBS for TAC-1.
Interface	A software entity which supports transmission of messages between two systems.
KB	Knowledge Base.
KB-BATMAN Shell	Knowledge-Based Battle Management Shell. The general-purpose framework of software with which domain-specific applications can be built to produce a Testbed.
KBS	Knowledge-Based System. Similar terms are expert system and intelligent system.
KNOBS	KNOWledge-Based System, a military mission planning system produced at MITRE-Bedford.
KRS	KNOBS Replanning System.
Message	A set of data transmitted from one input port to another.

Method	In Flavors, a behavior associated with a particular flavor (class of objects).
Mixin	In Flavors, a flavor (subclass) representing a well-defined set of features which is intended to be combined into other flavors.
Object-Oriented Programming	A style of software programming in which objects are manipulated. Objects can have state data (instance variables) and defined behaviors (methods).
PLANNER	The planning KBS for TAC-1.
RADC	Rome Air Development Center, Griffiss Air Force Base, Rome, New York.
RDB	Relational database.
RDBMS	Relational Database Management System.
Reply	A response to a particular request from a system.
Request	A declaration by a system of its desire to have a service performed outside the system. A reply is expected in response.
RI	Router Interface.
ROSS	Rand's Object-Oriented Simulation System language (McArthur, Klahr, and Narain, 1984).
Router	A process operating in the Testbed's subnet which directs messages to the appropriate systems.
Router Interface	A process operating in the Testbed's subnet which relays messages between the Router and a system interface.
Service	A capability that a system performs in response to requests from other systems.
SI	System Interface.
SIMULATOR	The simulation KBS for TAC-1.

Subnet	A centralized part of the Testbed which includes the Router and router interfaces.
Symbolics	A manufacturer of Lisp machine hardware and software.
System	A modular component which can be connected with the Testbed.
System Executor	A process which is triggered by a system interface to execute a requested service of a system.
System Interface	A process operating outside the Testbed's subnet which relays messages between a router interface and system executor processes.
TAC-1	The name of the first demonstration Testbed which is applied to the Air Force tactical battle management domain.
TAC-2	The successor to TAC-1.
TACC	Tactical Air Control Center.
TACS	Tactical Air Control System.
Testbed	A collection of systems and hardware built using the KB-BATMAN Shell which permits experimentation with different configurations of systems in a given problem domain.
T _E X	A document preparation system developed by Donald E. Knuth.
VAX	A line of computers manufactured by Digital Equipment Corporation.
VMS	An operating system for VAX computers.
WOC	Wing Operations Center.

MISSION of Rome Air Development Center

RADC plans and executes research, development, test and selected acquisition programs in support of Command, Control, Communications and Intelligence (C³I) activities. Technical and engineering support within areas of competence is provided to ESD Program Offices (POs) and other ESD elements to perform effective acquisition of C³I systems. The areas of technical competence include communications, command and control, battle management, information processing, surveillance sensors, intelligence data collection and handling, solid state sciences, electromagnetics, and propagation, and electronic, maintainability, and compatibility.

END

DATE

FILMED

9-88

DTIC